2006-12

Guidance and navigation software architecture design for the Autonomous Multi-Agent Physically Interacting Spacecraft (AMPHIS) test bed

Eikenberry, Blake D.

Monterey California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

GUIDANCE AND NAVIGATION SOFTWARE ARCHITECTURE DESIGN FOR THE AUTONOMOUS MULTI-AGENT PHYSICALLY INTERACTING SPACECRAFT (AMPHIS) TEST BED

by

Blake D. Eikenberry

December 2006

Thesis Advisor:	Marcello Romano
Second Reader:	Oleg Yakimenko

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 2006	Master's Thesis and Engineer Degree

4. TITLE AND SUBTITLE
Guidance and Navigation Software Architecture Design for the Autonomous Multi-Agent Physically Interacting Spacecraft (AMPHIS) Test Bed

5. FUNDING NUMBERS

6. AUTHOR
Blake D. Eikenberry

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited	A

13. ABSTRACT

The Autonomous Multi-Agent Physically Interacting Spacecraft (AMPHIS) test bed examines the problem of multiple spacecraft interacting at close proximity. This thesis contributes to this on-going research by addressing the development of the software architecture for the AMPHIS spacecraft simulator robots and the implementation of a Light Detection and Ranging (LIDAR) unit to be used for state estimation and navigation of the prototype robot. The software modules developed include: user input for simple user tasking; user output for data analysis and animation; external data links for sensors and actuators; and guidance, navigation and control (GNC). The software was developed in the SIMULINK/MATLAB environment as a consistent library to serve as stand alone simulator, actual hardware control on the robot prototype, and any combination of the two. In particular, the software enables hardware-in-the-loop testing to be conducted for any portion of the system with reliable simulation of all other portions of the system. The modularity of this solution facilitates fast proof-of-concept validation for the GNC algorithms. Two sample guidance and control algorithms were developed and are demonstrated here: a Direct Calculus of Variation method, and an artificial potential function guidance method. State estimation methods are discussed, including state estimation from hardware sensors, pose estimation strategies from various vision sensors, and the implementation of a LIDAR unit for state estimation. Finally, the relative motion of the AMPHIS test bed is compared to the relative motion on orbit, including how to simulate the on-orbit behavior using Hill's equations.

14. SUBJECT TERMS Autonomous on-orbit spacecraft assembly, fractionated spacecraft, LIDAR, navigation, proximity operations, multi-agent, robotic, hardware-in-the-loop test bed	15. NUMBER OF PAGES 147
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

GUIDANCE AND NAVIGATION SOFTWARE ARCHITECTURE DESIGN FOR THE AUTONOMOUS MULTI-AGENT PHYSICALLY INTERACTING SPACECRAFT (AMPHIS) TEST BED

Blake D. Eikenberry
Lieutenant Commander, United States Navy
B.S., University of California, Los Angeles, 1993

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ASTRONAUTICAL ENGINEERING
and
ASTRONAUTICAL ENGINEER DEGREE

from the

NAVAL POSTGRADUATE SCHOOL
December 2006

Author: Blake D. Eikenberry

Approved by: Marcello Romano
Thesis Advisor

Oleg Yakimenko
Second Reader

Anthony J. Healey
Chairman, Department of Mechanical and Astronautical Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Autonomous Multi-Agent Physically Interacting Spacecraft (AMPHIS) test bed examines the problem of multiple spacecraft interacting at close proximity. This thesis contributes to this on-going research by addressing the development of the software architecture for the AMPHIS spacecraft simulator robots and the implementation of a Light Detection and Ranging (LIDAR) unit to be used for state estimation and navigation of the prototype robot. The software modules developed include: user input for simple user tasking; user output for data analysis and animation; external data links for sensors and actuators; and guidance, navigation and control (GNC). The software was developed in the SIMULINK/MATLAB environment as a consistent library to serve as stand alone simulator, actual hardware control on the robot prototype, and any combination of the two. In particular, the software enables hardware-in-the-loop testing to be conducted for any portion of the system with reliable simulation of all other portions of the system. The modularity of this solution facilitates fast proof-of-concept validation for the GNC algorithms. Two sample guidance and control algorithms were developed and are demonstrated here: a Direct Calculus of Variation method, and an artificial potential function guidance method. State estimation methods are discussed, including state estimation from hardware sensors, pose estimation strategies from various vision sensors, and the implementation of a LIDAR unit for state estimation. Finally, the relative motion of the AMPHIS test bed is compared to the relative motion on orbit, including how to simulate the on-orbit behavior using Hill's equations.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS

AMPHIS	-	Autonomous Multi-agent Physically Interacting Spacecraft
NPS	-	Naval Postgraduate School
SSAG	-	Space Systems Academic Group
SRL	-	Spacecraft Robotics Laboratory
POSF	-	Proximity Operations Simulator Facility
DOF	-	Degrees of Freedom
CMG	-	Control Moment Gyro
MSGCMG	-	Miniature Single Gimbaled Control Moment Gyro
iGPS	-	Indoor Global Positioning System
GNC	-	Guidance, Navigation and Control
LIDAR	-	Light Detection and Ranging
QD	-	Quick Disconnect
OD	-	Outer Diameter
ID	-	Inner Diameter
NPT	-	National Pipe tapered Thread
RS-232	-	Recommended Standard -232
TCP/IP	-	Transmission Control Protocol/Internet Protocol
CPU	-	Central Processing Unit
LEO	-	Low Earth Orbit
I/O	-	Input/Output
ISA	-	Industry Standard Architecture
PCI	-	Peripheral Component Interface
USB	-	Universal Serial Bus
LPT	-	Line Printing Terminal
KVM	-	Keyboard, Video, Mouse
DRAM	-	Dynamic Random Access Memory
CMOS	-	Complementary Metal Oxide Semiconductor
SCBA	-	Self-Contained Breathing Apparatus

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

The author would like to acknowledge the financial support of the Naval Postgraduate School.

The author would also like to thank the following for their invaluable assistance in the completion of this thesis:

- My beautiful wife Carri and two wonderful children Grace and Jackson, who have kept me smiling despite the long hours away from home

- Dr. Marcello Romano and Dr. Oleg Yakimenko for their expertise and guidance throughout the thesis process

- LCDR Jason Hall, USN, and LT Bill Price, USN, for their friendship and partnership thus far in the developmental process of the AMPHIS test bed and for what they still will provide

- Captain Dave Friedman, USAF, and LCDR Tracy Shay, USN, for their discoveries of many best-practices in component selection and robotic vehicle construction

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The motivation behind the Autonomous Multi-Agent Physically Interacting Spacecraft (AMPHIS) test bed is the autonomous interaction of multiple, fractionated spacecraft. Many applications can be imagined for the ability of a spacecraft to interact with another spacecraft, including rendezvous for repair, refueling or replenishment, and salvage or rescue. The ability for multiple spacecraft to dock with one another is tremendously important because it would facilitate a new paradigm for putting satellites on orbit. No longer would a single, costly rocket launch be needed for all missions; instead, some missions could be launched in multiple smaller vehicles as independent units. Once on orbit, the units would autonomously maneuver and dock to form a single functioning satellite.

The AMPHIS test bed provides a platform to implement and test many different concepts of operations. Other applications can include multiple spacecrafts interacting with a non-cooperative target. Since these spacecraft are possibly already on orbit, it is not possible to presuppose the existence of any tell-tail features on the spacecraft, such as a pattern of special light emitting diodes (LED), lasers, radio frequencies, etc. The Russians have operated such systems for years using similar techniques, and of course, these pre-positioned articles greatly simplify the problem (Ref. [19]). A vision based approach could be taken to generalize the problem. Using a vision based approach, only general shape and size characteristics of the surrounding objects must be known in order to calculate bearing, distance and orientation. The AMPHIS test bed provides an excellent stage to develop solutions for these and other related problems.

A. BACKGROUND

The topic of autonomous interacting spacecraft is gaining popularity as space launch remains high cost, and automated space systems could be economical and beneficial for certain commercial and military operations. A few real-world projects of autononmy through the use artificial vision are given here for contextual background.

1. NASA's Demonstration for Autonomous Rendezvous Technology (DART)

The DART mission (FY05) was NASA's first test of an autonomous rendezvous in space. DART was supposed to demonstrate the technology needed to one day guide supplies to the ISS, service satellites on orbit, etc. It was equipped with an Advanced Video Guidance Sensor (within 330 feet) and Global Positioning System (GPS) (outside of 330 feet). Unfortunately, DART used more propellant than anticipated; when it tried to maneuver away, it struck the rendezvous satellite. (Ref. [19])

2. DARPA's Orbital Express

Orbital Express (FY06) will demonstrate enabling technologies for autonomous rendezvous, capture, serving, and maintenance of on orbit satellites. It will perform a series of captures and separations over various conditions. Electric and fuel coupling between orbital express and the rendezvous satellite will be tested, along with an onboard robotic arm to autonomously transfer several items. Visible and infrared artificial vision aids its autonomous capabilities. (Ref. [20])

3. National Space Development Agency (NASDA) of Japan's ETS-7

The ETS-7 program consisted of a pair of satellites, a chase satellite and a target satellite, that successfully undocked and re-docked autonomously in July of 1998. Also known as Kiku-7, the pair of interacting spacecraft performed multiple tests, including degraded equipment tests and several tele-robotic experiments that boosted Japan's hopes for future unmanned space flights. Scientists at NASDA claim that this experiment has proven the cost effectiveness of autonomous, interacting spacecraft. (Ref. [21]) However, since the system was built and launched together before it was tested, there is some doubt the system could be as effective with other types of spacecraft.

4. European Space Agency's (ESA) Automated Transfer Vehicle (ATV)

The ATV is a European developed spacecraft for providing the International Space Station with the automated transfer of supplies. The first of its class, the Jules Verne successfully completed the autonomous rendezvous and docking system in Europe's largest ship hull test facility in September, 2006 and plans to replicate that success in space in 2007. Its primary mode of navigation comes from the use of independent supervision laser scanning device. (Ref. [22])

5. Air Force Research Laboratory's (AFRL) XSS-10

The XSS-10 micro-satellite ejected from a Delta 2 rocket in January, 2003, and then maneuvered itself back to the spent stage. It repeated this sequence twice more before begin described as a success. Its navigation relied in part on an onboard television camera. The vision, propulsion and guidance and control software all performed well for the $100 million program. Its success is a key element in the development of future autonomous spacecraft. (Ref. [**Error! Reference source not found.**])

6. AFRL's XSS-11

The XSS-11 Demonstration Mission was launched in April of 2005. Its purpose was to demonstrate robust, extended duration proximity operations. It is a micro-satellite class vehicle that could autonomously rendezvous with multiple space objects using a scanning LIDAR for navigation. It also has several guidance modes, such as forced-motion trajectories, closed loop proximity operations, or collision avoidance that could be switched from ground control or autonomously. (Ref. [24])

7. NASA's Hubble Robotic Vehicle (HRV)

NASA was developing a robotic vehicle to service the Hubble Space Telescope (HST) in FY08. Its purpose was to lengthen the life of the HST by taking it new batteries, propellant inside of a de-orbit module, and an ejection module with robotic units. The HRV would be paritially controlled from the ground to install new instuments, and reroute power using the new batteries. The HRV project was recently cancelled due to budget constraints. (Ref. [25])

8. Obital Recovery Group's Orbital Life Extension Vehcile (CX-OLEV)

CX-OLEV's mission is to prolong the lives of telecommunications satellites. CX-OLEV will operate as space tug, carrying the propellant, and navigation to boost telecommunications satellites into the proper orbit, extending their life by approximately eight years. It will dock with the rendezvous satellite's apogee kick motor using artificial vision. Over one hundred satellites have been identified that could benefit from CX-OLEV. The first mission is scheduled in 2008. (Ref. [26])

9. Naval Research Laboratory's (NRL) Spacecraft for the Universal Modifications of Orbits (SUMO)

The SUMO project at the NRL (currently being renamed to FREND) is being developed to be somewhat of a space "AAA truck." It is a satellite with a hefty fuel (ΔV)

capacity and multiple robotic arms. If a satellite becomes incapacitated due to malfunction, runs out of fuel, or just does not have the fuel needed to change to the desired orbit, the FREND craft will maneuver up the target craft and grab onto it with its robotic arms. It will then use its own propulsion to move the target craft into the desired orbit. The NRL also operates a six degree of freedom, on orbit simulator. The simulator takes actuator input and manipulates the satellite with the appropriate dynamics in all six degrees of freedom, including the motion created by differing orbits. (Ref. [27])

10. AUDASS at SRL of Naval Postgraduate School (NPS)

The Spacecraft Robotics Laboratory (SRL) at the Naval Postgraduate School supports the Graduate School of Engineering and Applied Science (GSEAS), the Space Systems Academic Group (SSAG), and conducts research for the Air Force Research Lab (AFRL) (Space Vehicle Directory), Defense Advanced Research Projects Agency (DARPA) (Tactical Technology Office), and various sponsoring agents. The first interacting spacecraft simulator robot project conducted in the SRL at NPS was the Autonomous Docking and Servicing Spacecraft Simulator (AUDASS). (Ref [1], [3], [4])

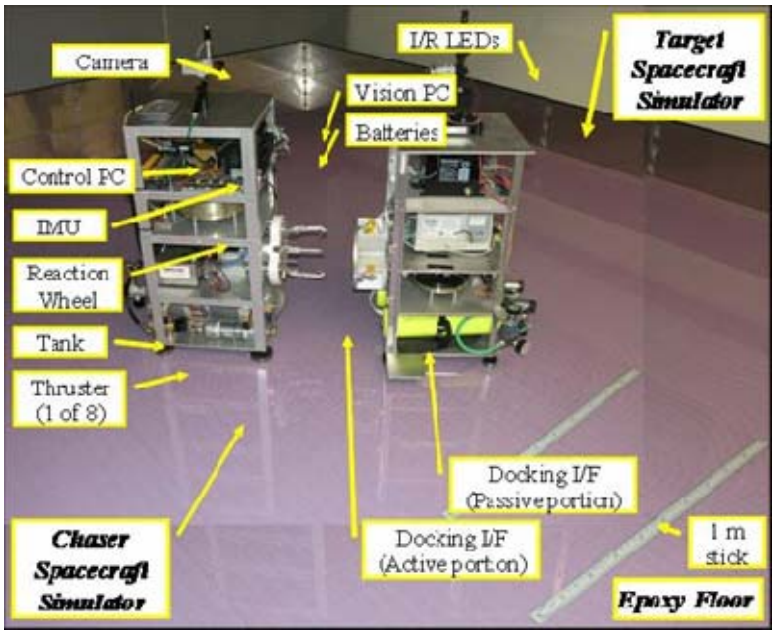


Figure 1. Autonomous Docking Testbed at the NPS SRL (Ref. [1])

This project first conducted an autonomous docking on the planar floor in the fall of 2005. For reference, each robot had the same basic physical properties: mass of 63 kg,

moment of inertia about the vertical axis of 2.3 kg m^2, maximum control torque about vertical axis of 0.16 Nm, and maximum thrust of 0.45 N per thruster. The vision sensor used was a monocular camera which produces a two dimensional image that determined its relative position by focusing on three non-planar infrared lights positioned on the target robot. (Refs. [1], [2], [3], [4])

Contrary to the laboratory simulator at the NRL, the simulator at SRL can only simulate three degrees of freedom, vice six. But the differences between them create a completely complementary set of labs: the simulator at the NRL cannot test with the actual actuators in the loop. Although the SRL can only test three degrees of freedom, every element, from sensor to actuator can be tested in the hardware loop.

B. PROJECT GOALS AND LIMITATIONS

Development of the AMPHIS project represents the realization of several overarching goals. It provides a platform for graduate level academic learning as it is an unsolved, real-world problem requiring expertise in multiple disciplines. As an engineering problem, it contains many foreseen and unforeseen challenges that require innovation and cooperation from a dedicated team of engineers. Therefore, once developed, the AMPHIS test bed will serve as a platform for simulation, development, implementation, testing and evaluation for different sensors, actuators, artificial visions, and guidance, navigation and control algorithms to validate and perfect a solution for the multiple interacting spacecraft problem.

The AMPHIS project has several key limitations in scope:

1) The spacecraft simulator robots have only three degrees of freedom (3 DOF): translation on a flat, level plane, and yaw – the rotation about the vertical axis. Implementing a hardware system that simulates a gravity free, frictionless environment for a "free floating" robot with pitch, roll, and yaw is beyond scope of the project. However, a multiple spacecraft, six degree of freedom (6 DOF) computer simulation with multiple perturbations can be used to simulate the full problem (Ref. [5], [6], [8]). Also, the applicability of a 6 DOF system using a 3 DOF simulator is discussed in Section IV.

2) The part selection is also limited, in general, to commercial off the shelf (COTS) items; the only fully contracted part is the specially made, frictionless floor which is flat to a high degree of accuracy. The limitations on parts sometimes require inefficient, inelegant work-a-rounds; at the same time, they also present unforeseen opportunities to innovate and engineer solutions to difficult challenges. These problems include: portable processing speed and capacity limitations; hardware device communication and synchronization, and sensor translation and integration. These limitations will be clarified further in the next section.

C. EXPERIMENTAL SETUP

The major components of the AMPHIS project are the Proximity Operation Simulator Facility (POSF), and the spacecraft simulator robots. The POSF consists of a special flat floor that measures 4.9 m by 4.3 m. Its surface is made of Epoxy material. When used in conjunction with the air pads, the floor is essentially frictionless and horizontal to a high degree of accuracy (residual gravity ~10^{-3} g) (Refs. [1], [3], [4], [5]).

The spacecraft simulator robots float via air pads over the floor. Each robot has three degrees of freedom, two for the translation and one for the rotation about vertical axis. Although one of the goals of the AMPHIS project is to test different sensors, actuators, and equipment, each robot must have certain elements to function correctly, including:

- Air pads, to reduce the friction of the POSF
- Thrusters, to provide translational movement on the planar floor and rotation
- Compresses air system, to operate the air pads and thrusters
- Reaction wheels, or control moment gyros (CMG), for attitude control
- Gyros, to sense changes in attitude
- Accelerometers, to sense translational movement
- On board computers, to calculate and control the hardware devices
- Wireless adapters, to communicate with other robots

- Indoor GPS, to sense an absolute position in the room
- Line counters, to determine movement on the POSF by counting lines on the POSF (not yet developed at SRL)
- Laser mice, to determine movement on the POSF by sensing movement of the mouse (not yet developed at SRL)
- Artificial vision, to determine the positions of other robots and obstacles on the POSF
- Docking mechanism, to dock with other robots
- Battery and power distribution system, for powering all electric devices onboard

Furthermore, the AMPHIS project expands testing past docking, so three spacecraft robots will be constructed once the first prototype has been developed.

Although the AMPHIS test bed will have the ability to test multiple design concepts, a base concept was decided on for initial implementation. This base concept consists of the following equipment on each robot:

- 2 Batteries and power distribution system
- 4 Air pads
- 4 tank compressed air system
- 2 thrusters (front and aft) that rotate ±180˚
- 2 micro CMGs

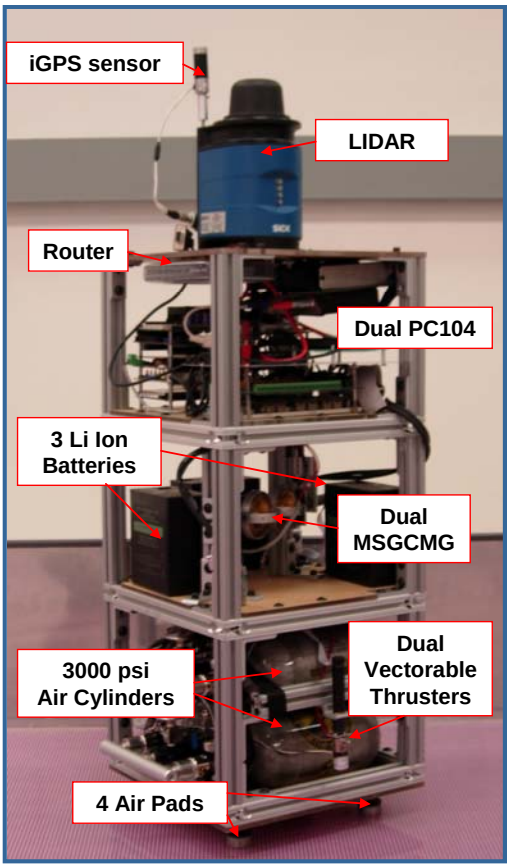


Figure 2. Base concept configuration (Ref. [5])

- Indoor GPS receiver
- 1 Gyro
- 2 Accelerometers
- 2 onboard PC104 computers
    - One running xPC Target in real time for guidance, navigation and control
    - One running Windows XP for GPS and LIDAR processing
- 1 Wireless adapters
- 1 Ethernet router for connectivity
- 1 Sick LIDAR OEM

Figure 2 is a photograph of the prototype simulator robot in this configuration, and an illustration of the concept of operations is depicted in Figure 3.

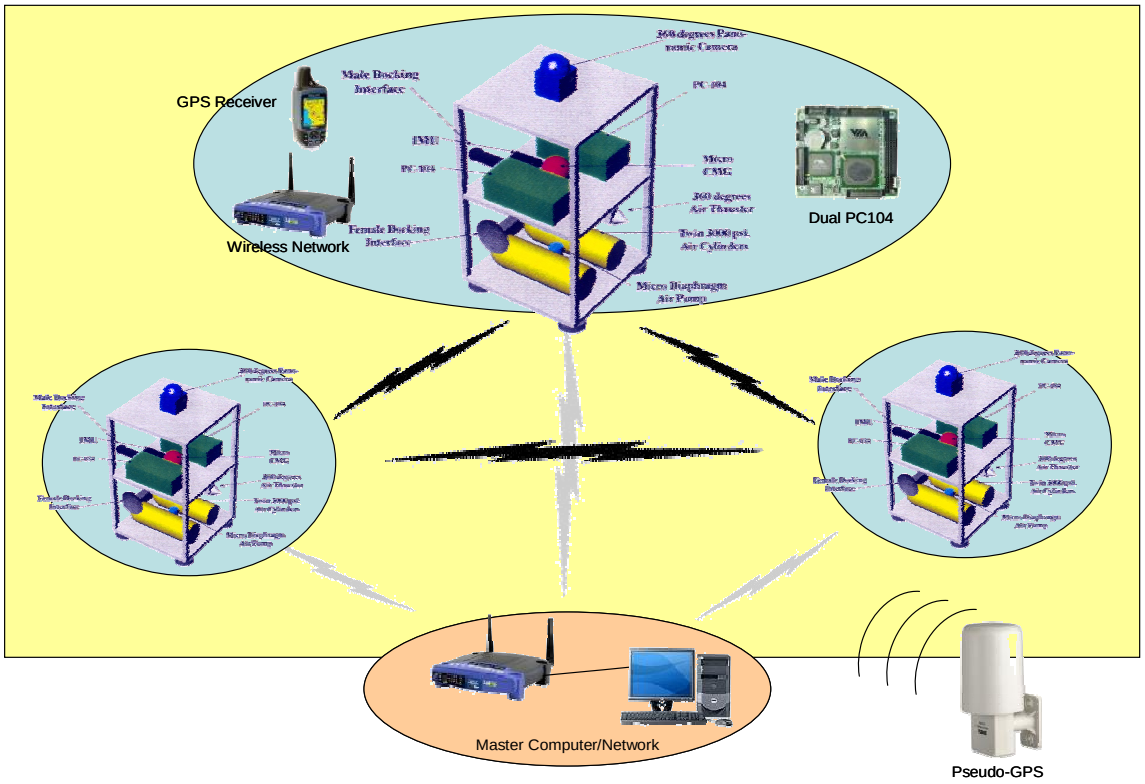


Figure 3. Experimental Setup

Each robot will be equipped with the Indoor Global Positioning System (GPS) and a mono-vision camera. The Indoor GPS system acts similarly to GPS within the laboratory and is composed of two stationary emitters and one receiver on each robot. These units are calibrated so the sensor on each robot can determine its position on the floor to within a few millimeters. The LIDAR mounted on the prototype unit is used to find the positions of the other robots relative to it. Details of the SICK OEM LIDAR are presented in the Section III. Communications between robots via a wireless network will be integrated into the system. The onboard computers handle image processing for state estimation, compute control profiles, command thrust and torque actuators and are linked to a wireless network for data exchange among the robots. The wireless network enables

multiple cooperation paradigms. The hardware construction of these robots has been detailed in References [1], [3], [4], [5], [6], and [7]. A detailed test bed schematic is displayed in Figure 4.

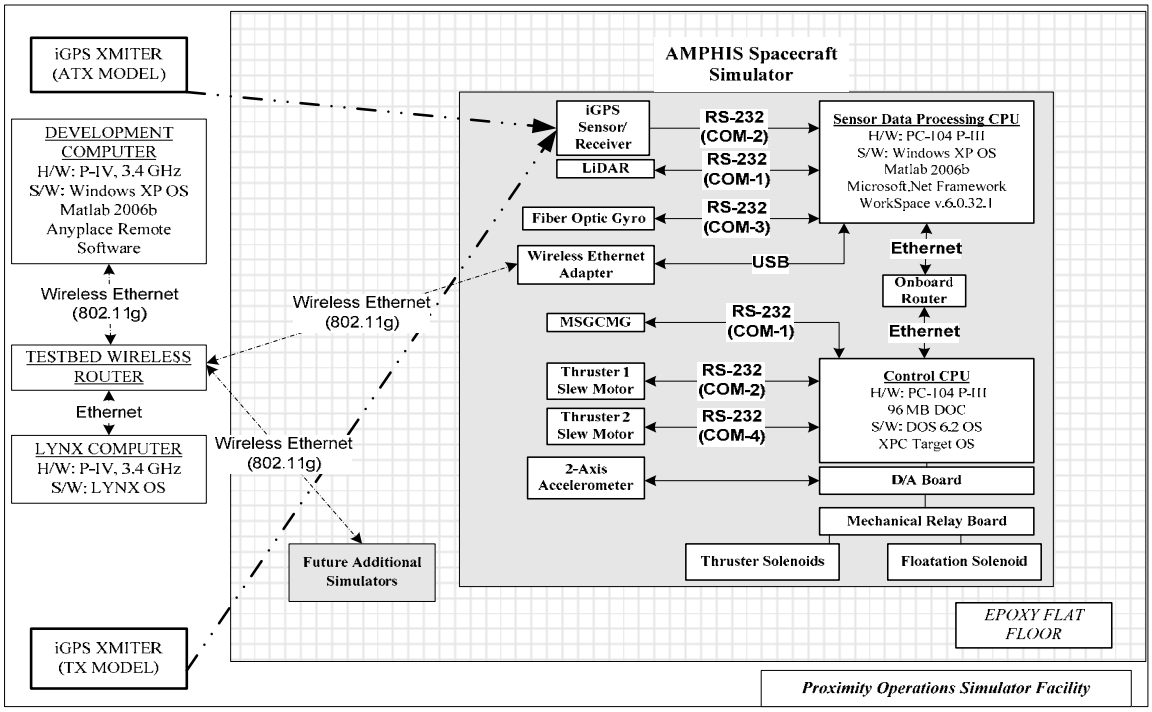


Figure 4. AMPHIS Test Bed Schematic (Refs. [5], [6])

Hardware limitations to this point have prevented a fully functioning proto-type. The only way to use the Indoor GPS system in real time is by using the manufacturer's program, called Work Space, which must run on Windows XP. The Work Space protocol cannot yet run in MATLAB, but only in LINUX. Therefore, the information flow must go from the GPS transmitters to the GPS receiver, to the Work Space program on the onboard Windows XP PC104, to an off board LINUX computer via the wireless TCP/IP Local Area Network (LAN) for processing, back to the PC104 computer via TCP/IP LAN, and then finally to the xPC Target PC104 via UDP. The processed information cannot be sent directly to the xPC Target computer because xPC target computer is not compatible with any wireless adapter. Not only have these hardware limitations created en inefficient implementation of the design concept, but the PC104 processor has not been able to handle the workload: Windows XP, Work Space, and MATLAB/SIMULINK cannot all run concurrently and flawlessly.

II. SOFTWARE DESIGN

Requirements for the AMPHIS software design were based much upon the concept that a single program could be used in multiple places in various ways without having to be completely re-engineered to function. For example, the code needed to function as a stand alone simulator, yet it also had to compile into xPC Target code to function as the guidance, navigation and control program on the spacecraft simulator robot. It needed to function for one, two, or three robot scenarios. It also needed to be installed on multiple robots without the need for a lot of customization. Fulfilling these requirements simultaneously make the design much more difficult than intuition says it should be. The section will explain how some of these design challenges were met.

A. USES OF THE SIMULINK/MATLAB SOFTWARE

There are two computers in the base design of the AMPHIS spacecraft simulator robots. As previously mentioned, one computer will run Windows XP for wireless device transmissions and for LIDAR processing. The SIMULINK model for the Windows computer will be held to the end of the section. First, the software for the guidance, navigation, and control xPC Target machine will be discussed.

The first principle concepts in the software architecture that facilitates meeting all of the aforementioned requirements is that the software is designed around the functioning of a single robot which can "sense" the other robots. This approach differs from a design that treats all robots as equals, as a simple simulator would do. As the primary robot, or the robot of focus, the software has guidance, navigation and control only for itself; the state (position and velocity) of the other two robots is simulated from a user defined profile, sensed with artificial vision, communicated via the wireless LAN, or calculated from a combination of several inputs.

Since a single code is needed to be deployed on multiple robots, and then communicate together, a naming scheme is needed to prevent conflicts between functioning robots. For example, if each physical robot was named Robot 1, Robot 2, and Robot 3 (denoted as uppercase "Robot"), it is desired to employ the single software code on each robot without having to rename all of the internal variable names (denoted

as lowercase “robot”) to coincide with the global naming convention. For this reason, a relative-naming convention was contrived to limit the reconfiguration to the setting of a single variable, referred to as simply the identification (id#). Since the single code is focused on the robot to control, the relative name for this robot is Robot 1. In other words, robot 1 refers to all things having to do with the controlled robot, and the other Robots are referred by as robot 2 and robot 3. Since this the naming convention could become confusing, this translation matrix is displayed on the top level of the SIMULINK model:

Id# / internal name	robot 1	robot 2	robot 3
Id#1	Robot 1	Robot 2	Robot 3
Id#2	Robot 2	Robot 1	Robot 3
Id#3	Robot 3	Robot 2	Robot 1

Table 1. Relative to global naming translation

This matrix is interpreted as such: the numbers in the matrix indicate the global names, or hardware names of the robots. They could be interchanged from 1, 2, 3, to A, B, C; Blue, Red, Green; Huey, Dewey, and Louie; etc. Since each robot is assigned a different identification number, it is used in conjunction with the interpretation matrix the match relative names with global names. The id# determines which row the software will index the naming scheme. Then, indexing the columns using the relative, internal names (lowercase “robot” tags used in the universal software) will give the global name of a particular robot. For example, for id#1 (the software running on Robot 2), the internal name “robot 2” refers to Robot 1, and “robot 3” refers to Robot 3. Similarly, for id#3 (the software running on Robot 3), the internal name “robot 2” refers to Robot 2, and “robot 3” refers to Robot 1. With this naming convention, the internal name “robot 1” will always refer to its own global name. Note also that Robot 1 and Robot 3 both refer to robot 2 as Robot 2. This naming convention allows for the portability of the single control code to any three robots. By setting the id# uniquely, the data will be indexed correctly for that particular robot.

One key concept for the architecture design of the software design is the simulator/control software duality. To be truly useful for hardware in-the-loop testing,

the design needed to not only function as either a simulator or a control platform, but also as a hybrid, controlling some things and simulating others. Simple manual switches were placed in key areas to facilitate user ease of selecting what needs to be simulated and what needs to work as a controller or sensor. These key areas are:

- The state of robot 1
    - Onboard sensors of robot 1
    - The plant model (state integrator)
    - Simulated from user defined trajectory lookup tables
- The state of robots 2 and 3
    - Onboard sensors of each robot (via UDP)
    - Simulated from trajectory lookup tables (user defined)
    - Vision sensors (LIDAR)
    - Simulated vision sensors (simulated from the trajectory lookup tables)

Depending on the set of sensors used, it may be desired to use a combination of several sensors, rather than just one. A Kalman filter can provide real time updates even though updated from the vision sensors will happen at discrete intervals in non-real time.

The system described above can facilitate many simple configuration changes for different testing scenarios. This key concept is an important factor when trying to develop one or more of the modules in Figure 5. Each of these blocks can be operated in simulation mode, or control mode.

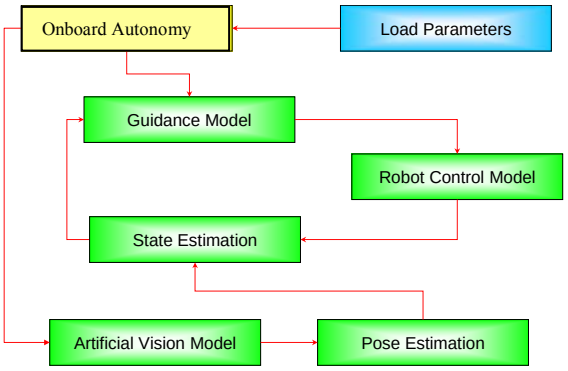


Figure 5. xPC Target Software hierarchy (Ref. [17])

Since the state of any robots is easily simulated with user defined trajectories using lookup tables, robots can be tested one at a time. By simulating the other one or two robots, the robot of focus can be developed, testing the guidance, navigation and control blocks. To more accurately model the artificial vision sensor, parameters on the pose estimation simulator can be configured to match realistic update rates and accuracies.

Another point key to the successful implementation of the control/simulator software is in the ability to conduct hardware in-the-loop testing. Regardless of which modules are being simulated, they can all be compiled and run in xPC Target. This way, the control actuators can be tested without the robot actually having to move. As the coordinates and attitudes of all the robots can be simulated, the actuator control and feedback signals can be viewed and examined to the state trajectories given to it.

B. PROGRAMMING PRACTICES AND RULES USED IN THE SOFTWARE DESIGN

Several programming practices had to be employed to ensure several requirements could be met. First, the software had to compile and run on xPC Target. Second, the architecture had to enable blocks from multiple designers to be integrated seamlessly into the master software. Third, the architecture had to support multiple guidance, navigation and control schemas. For example, a trajectory planner guidance system works significantly different than an artificial potential function guidance system. The architecture is desired to be flexible enough to have the major modules designed and

developed without having to redesign other modules. Finally, the architecture had to support growth: new and different equipment will need to eventually be incorporated into the design.

Several basic, good programming practices had to be followed to ensure the aforementioned requirements could be met. These practices are listed and explained here.

- Modularize the design. A modular design is instrumental in the facilitation of the fulfillment the above requirements. Since there are many ways to implement solutions to any given problem, and multiple ways to define the division of labor of subsystems, a clear definition of the purpose, functionality, and interface of each module must be made to ensure development from different designers share the same vision of the overall architecture. These explanations follow in the next several sections of this paper.
- Standardize the interface between modules. The interface between modules, such as the format, size, name, and context of the inputs and outputs of every module, must be defined to facilitate integration of different modules from different designers. Again, these explanations follow in the next several sections of this paper.
- Avoid global tags. Global tags are often used to prevent "spaghetti" code when sending signals from a subsystem to one or more different subsystems in SIMULINK models with "From" and "Goto" blocks. But there are problems with global tags: global tags take more processor time as seen when analyzing a system with the optional SIMULINK Optimizer. (Ref. [7]) Global tags also can lead to confusing models; all of the inputs and outputs can not be seen from the top level system making the requirements of the system function misleading. Instead of using global tags, the consist use of local tags with buses alleviate all of the above issues.

- Use bus creators and bus selectors where applicable. A bus is a wonder tool compared to the standard "MUX" and "DEMUX" blocks for many applications. When needing to send many disparate signals from one subsystem to others, a bus selector can incorporate many different signals onto one line. Furthermore, each line can be named for easy identification on the bus. When using the bus selector, it is only necessary to select the outputs needed for each subsystem; on the contrary, use of the MUX and DEMUX blocks require the entire line to be broken apart in every subsystem it is utilized. This feature allows the sizing requirements for signals to be kept consistent much easier to with buses; therefore buses facilitate seamless integration of modules from multiple designers. As equipment is added, removed, or changed, the data routed to the bus creator can easily be altered to accommodate the new signals without affecting the bus selectors on the other end; only the subsystems that use the new data will need to be changed accordingly. Buses have also shown to take less processing time compared to MUXes from the SIMULINK Optimizer. (Ref. [7])

- Avoid global variables. Similar to global tags, global variables can cause naming problems. Besides the burden of ensuring a global variable is correctly declared as such within every scope it is used, it becomes difficult to track many different global variables when debugging, and naming problems could cause conflict when one designer uses a global variable name, and then their code is integrated with code in which another designer used the same name for local variables. Also, embedded MATLAB functions do not allow the declaring of global variables. One important exception is found in the SIMULINK code for the Windows computer. Since objects cannot be passed in SIMULINK, the variable which contains the serial port information for the LIDAR connection must be declared as global.

- Avoid enabled subsystems. Enables subsystems are subsystems that operate only with they are enabled by some input parameter. Enabled subsystems cannot contain rate transition blocks (nor can they contain global tags, but those are avoided as well). As the AMPHIS project develops, some hardware may need these rate transition blocks to function correctly. If rate transition blocks are required in an enabled subsystem, a re-design would be necessary to incorporate both features. The work around for this conflict is not simple and case dependent.
- Avoid MATLAB Function blocks. MATLAB Function blocks are not fully compatible with xPC Target, and therefore cannot be used on the xPC Target machine. In the non-real time Windows computer, however, one is used for LIDAR control until the design can be ported over to xPC Target. Where applicable, Embedded MATLAB Function blocks are a better substitute and are used.
- Use a standardized naming convention. Using standardized names are less of a problem when almost all variables are locally defined. Only the naming interface between modules needs to be standardized. Primary name standards are as follows:
    - state: refers to the system state; for the base configuration, these variables are the coordinates and attitude (x_i, y_i, θ_i), and the time rate of change of x_i, y_i, θ_i for i = 1 to 3 robots.
    - state 1, or st1: refers to the state of robot 1, or the robot on which the software is running.
    - dead reckoning, or dr: the state as calculated only by the plant propagator.
    - ref, or user: the user defined reference signal, containing the desired end state
    - gcmd: guidance commands for the guidance module
    - vcmd: vision commands for the vision sensor

- act: signals for the actuators
- act_fb: actuator feedback signals
- xlink: the crosslink between robots, for incoming and outgoing messages
- input_bus: all of the information from external connections needed for state estimation
- u1: a structure containing all control related variables for robot 1 to be saved for animation and analysis
- v1: a structure containing all vision related variables for robot 1 to be saved for animation and analysis

C. USER INPUT

The only input that is desired to get from the user is the end state, or end position of each of the robots. An initial state must also be specified, but in the case of state determination from onboard sensors, it is desirable for each robot to determine the initial state autonomously. Of course, a simulation will always require a specified initial state. The desired end position can be expressed in two different ways: absolute and relative. Every maneuver considered here will be a rest to rest maneuver, so the discussion of initial and final rates will be limited to say that they are all zero.

An absolute end state is the simplest and most forward way to express the desired outcome of the maneuver. This method is simply defining the final coordinates and attitude for each robot. For example, an absolute end state could be:

	X_f	Y_f	Θ_f (attitude)
Robot 1:	2.0 m	3.0 m	10°
Robot 2:	1.5 m	2.0 m	95°
Robot 3:	4.0 m	3.5 m	190°

Table 2. Sample absolute end state

Note that the user is not inputting how to arrive at the end state; the onboard guidance, navigation and control systems will have to successfully maneuver each robot, with collision, to the final desired positions.

Since a large part of the AMPHIS project deals with relative motion, the other way to define the final end state is in relative coordinate. This can be accomplished by defining six variables (per robot) that describe a desired end state: a relative bearing to each robot, a range to each robot, and an angle that defines the orientation of each robot on its relative bearing. Note that there is no absolute information defined here. The system must find absolute values itself for the described system. For example, Figure 6 is a sample desired end state and the corresponding matrix from Robot 1 (blue). Again, the final desired rates are always zero.

From Robot 1 (blue)	Range	Bearing	Orientation
To Robot 2 (red)	R_{12}	B_{12}	O_{12}
To Robot 3 (green)	R_{13}	B_{13}	O_{13}

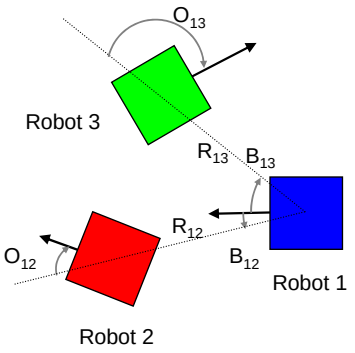


Figure 6. Relative parameters defining three-robot formation

Each robot must have a similar set of relative values that define the equivalent formation: the numbers that describe the final position relative to Robot 1 will be different for Robot 2 and Robot 3; however, they will all be related. For ease of user input, there is an automatic translation for relative final end states. This translator is the simplest of the three major modules on the top level of the SIMULINK model (upper left module in Figure 7). Once the matrix in Figure 6 is created from the perspective of Robot 1, the translator will put the matrix in a format from the perspective of the robot's identification number. This geometrical transformation is included in the Appendix.

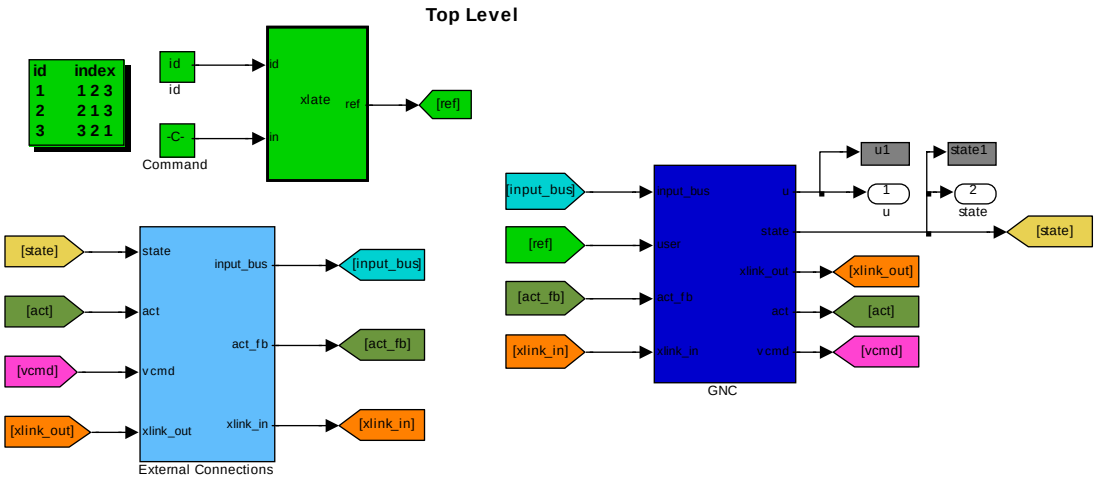


Figure 7. Top level of the xPC Target SIMULINK model

D. USER OUTPUT

There are three major categories of data that is desired to be output to the user for analysis and animation. These are: state data, control data, and vision sensor data. To prevent a convoluted workspace, an output structure variable is used via a simout block for each of these categories, named "state", "u", and "v" respectively. All values needing analysis are saved in one of these three variables. For xPC Target, the out block is used at the top level of the model alternatively, because xPC Target does not support the simout block. (Ref. [6]) With the amount of moving parts in the system, it is imperative to enable animation of the simulations to be able to have an accurate sense of what is happening. Specifically, it is necessary to follow the movements of each robot and relevant moving parts, such as vectored thrusters, throughout the maneuver. Figure 8 displays a single frame of a simulation using three robots in the base configuration.

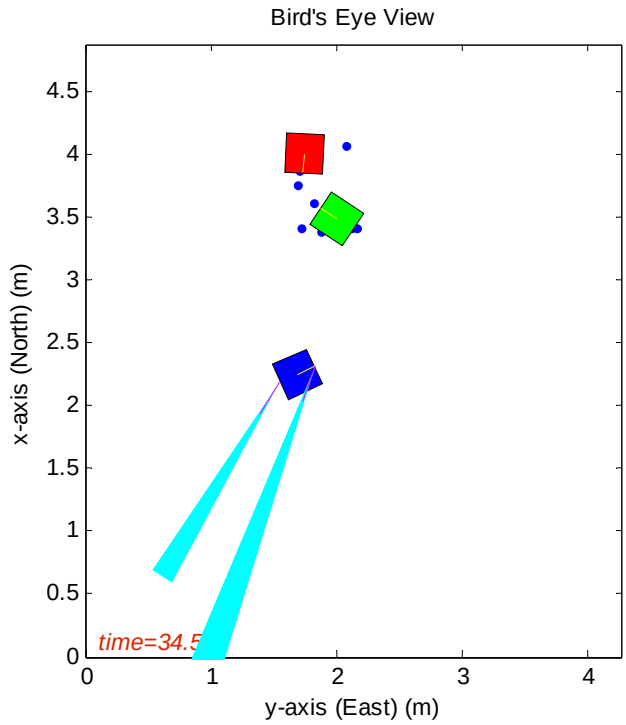


Figure 8. A single frame from the Bird's eye view animation

Note the dual fore and aft vectored thrusters are shown with their relative pointing direction on the blue (lower right) robot, and also their amount of thrust produced at a given instant indicated by the plume coming out of it. Successive frames plotted in this manner facilitate a clear understanding of the strengths and weaknesses of control scheme.

Figure 9 illustrates another perspective of animation developed for a slightly different configuration (see the section on "Direct Calculus of Variation Method" for a full description of the configuration). In this configuration, there is a single camera on each robot that has the capability to slew 360°. A view from any of the three cameras can be simulated at and animated as illustrated. On the left Figure 9, a bird's eye view of the floor is shown highlighting the field of view from the blue (lower left) robot's camera. The right side of Figure 9 depicts what is seen in that field of view from the camera perspective.

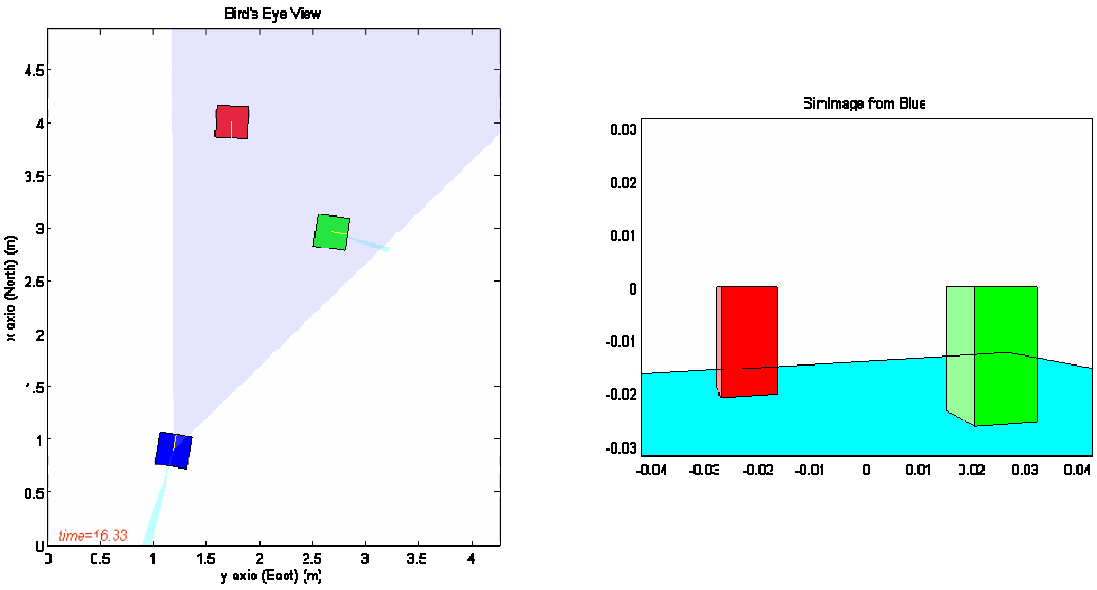


Figure 9. (Left) Bird's eye view showing the field of view, and (Right) the corresponding simulated photograph from the camera on the lower left robot

An animation script can automatically be called upon the completion of the simulation. This script is customized to animate the devices for a particular configuration and uses the three structure variables: state, u, and v. This animation code is included in the appendix.

E. EXTERNAL CONNECTIONS AND DATA LINKS

The External Connections and Data Links module is located in the lower left in Figure 7. The purpose of this module is to have all external connections collocated to facilitate easy reconfiguration and debugging of external devices. The three categories of connections are the Wireless LAN, Onboard Sensors, and Actuator Feedback. The External Connections SIMULINK module is shown in Figure 10.

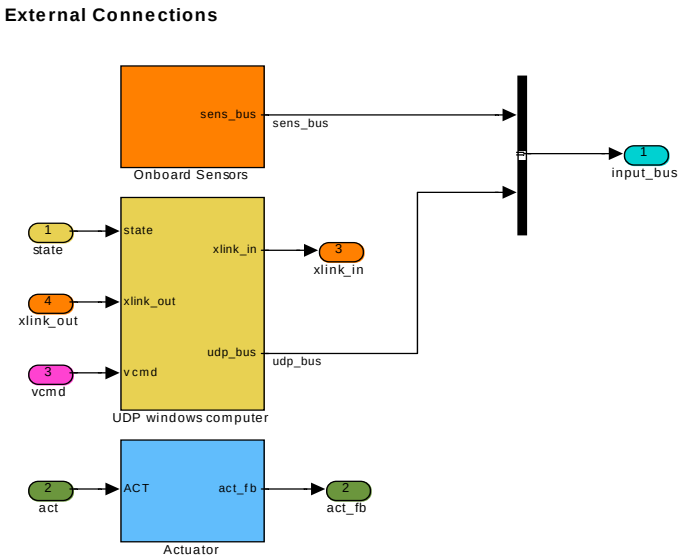


Figure 10. External Connections SIMULINK module

1. Onboard Sensors

All onboard sensors and actuators are connected via RS-232 serial cables, as most devices are commonly available with RS-232 connectors, and more importantly, xPC Target has ready made blocks to interact with RS-232 devices. The ports associated with these connections are commonly known as COM1, COM2, COM3, and COM4. In the base configuration described in a previous section, the onboard sensors that are directly interfaced through xPC Target are the gyro, and the accelerometers. Indoor GPS and vision sensors are interfaced indirectly through the onboard Windows XP computer, as explained in the following section. All of the data collected from the onboard sensors can be used to help with state estimation. Figure 11 displays the blocks used for the onboard sensors.

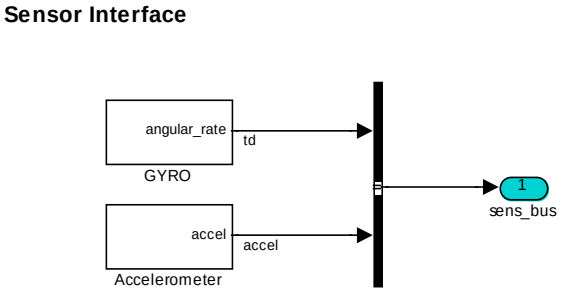


Figure 11. Onboard sensor RS-232 connections in SIMULINK

Refer to References [5], [6] and [7] for full discussion on sensors.

2. Actuators

The Actuator module collocates all of the actuator connections in one module. In the base configuration described in a previous section, the actuators that are directly interfaced through xPC Target are the micro CMGs and the thrusters. Actuator feedback is a collection of signals returned from the actuators that are used to ensure that the actuators are reacting properly to the given control signal. For example, the actual positions of the thrusters and CMGs are part of the actuator feedback. The Control algorithm will use this information by comparing it to commanded positions of the actuators. Refer to References [5], [6] and [7] for full discussion on actuators.

Actuator feedback interface is similar to the onboard sensor interface. Both are connected via serial ports and are interfaced directly with xPC Target. The major difference between the two is where the data is sent and how it is used: the onboard sensor data is sent to the Navigation module for state estimation, while the actuator feedback is used only in the Control module. Figure 10 illustrates how the actuator inputs and outputs are held separate from the other external connections.

3. Wireless Local Area Network (via the Windows XP Computer)

The Wireless Local Area Network (LAN) is for communication between robots. In a cooperative robot scenario, it is useful to have information sent between robots. State information passed between robots provides a very robust sensory network, especially when compared to onboard vision sensors. Other things, such as guidance modes and status messages are also useful to pass for coordination and synchronization between cooperating robots. In Figure 10, these messages are indicated by the "xlink_in" and "xlink_out" tags. The xlink messages are generated and utilized by the Navigation Module. Refer to the Navigation chapter for a more detail explanation on this topic.

A signal for artificial vision sensor control is also provided in the external connections module (indicated as "vcmd" in Figure 10). The Windows XP computer provides artificial vision sensor control as described in the next section; it therefore needs to have a communication link from the Navigation module that controls vision commands. In the case of a LIDAR, these commands are simply to activate or stop

LIDAR sensing and processing. A different sensor, however, may require more complicated control signals. A rotating camera, for example, may require interactive pointing commands.

Communication between the xPC Target and Windows XP computer is accomplished using User Datagram Protocol (UDP). Blocks for sending and receiving data using this protocol are predefined in xPC Target. The only parameters required to setup these blocks are:

1) IP address
2) Port number
3) Maximum data packet size

Figure 12 displays the UDP send and receive blocks for communication with the Windows XP computer. The entire IP address and port number configuration table is displayed in Table 3. The connections to the off board LINUX are discussed in the following section.

IP Address (192.168.)

Device	Robot1	Robot2	Robot3	Shore
ETHERNET (.1.)				
Router	111	211	311	
Windows	112	212	312	
xPC	113	213	313	
WIRELESS (.2.)				
SSID	heweynet	leweynet	deweynet	amphisnet
Router	111	211	311	1
Windows / Linux	112	212	312	10

Port Numbers

		FROM						
		Win1	**Win2**	**Win3**	**xPC1**	**xPC2**	**xPC3**	**Linux**
TO	**Win1**		5021	5031	4001			5000
	Win2	5012		5032		4001		5000
	Win3	5013	5023				4001	5000
	xPC1	4002						
	xPC2		4002					
	xPC3			4002				
	Linux	GPS	GPS	GPS				

Table 3. IP address and port number configuration

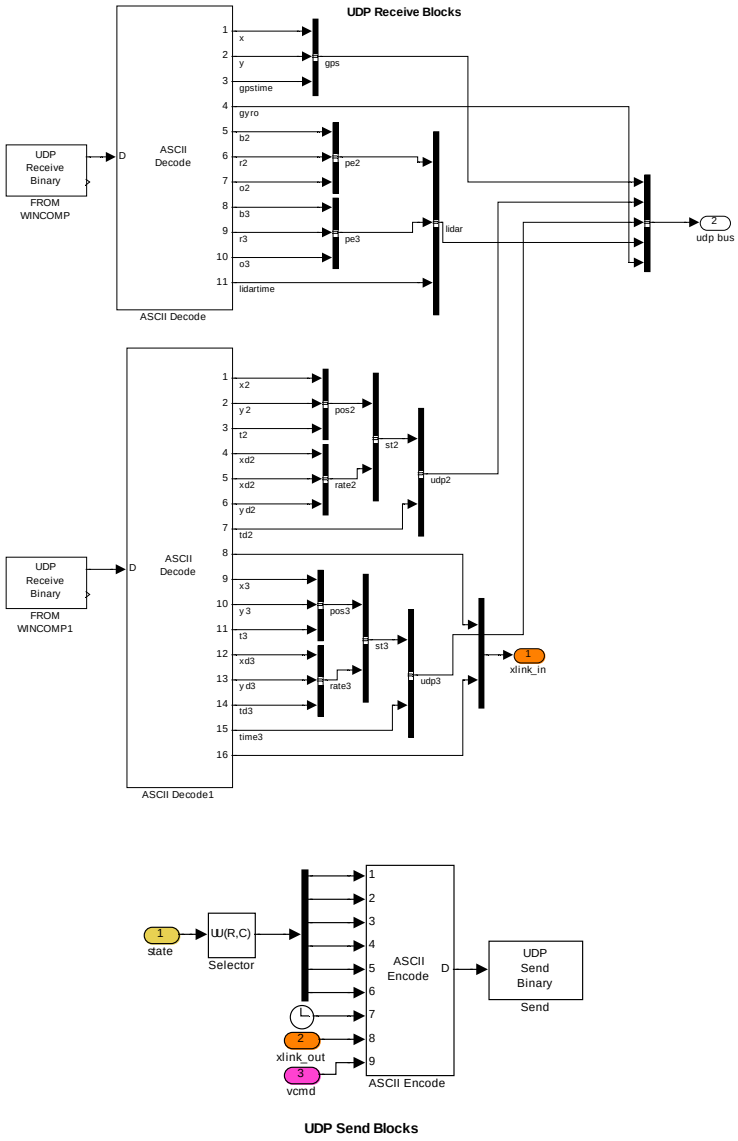


Figure 12. UDP send and receive blocks for communication with the Windows XP computer

The bus creator in Figure 10 collects all of the date for state estimation from the Windows XP computer and from the onboard sensors, and routs it to the Navigation block. Refer to the Navigation chapter for an explanation how this data is used.

F. GUIDANCE, NAVIGATION AND CONTROL

The SIMULINK guidance, navigation and control (GNC) module is the code that takes the system from its initial conditions and then, based on sensor input, manipulates the actuators in a way to move the system to the desired final state. It is seen on the right

of Figure 7. The basic interaction between the three parts: guidance, navigation and control, follows. The navigation module provides two functions. First, it uses sensor information to determine the system state. It then uses that information to manage how the guidance system will operate. For example, if the robots are separated by a large distance, each robot may use a different guidance mode than if they were closely separated and ready to dock. The guidance module will then take into account the current system state and the final desired system state and task the control system to move in the manner decided. The control module then takes its task and feedback from the actuators to calculate the control inputs for the actuators. As the actuators move the robot, and therefore change the system state, the navigation updates its estimation for the state, and the cycle continues. Each module will now be discussed in more detail. It is important to state that this discussion is based largely on the base configuration previously described; moreover, the behavior of these modules will depend on how the designer implements them. There is no single correct answer, and therefore, there is no standard way each of these modules will interact. This discussion will therefore remain general, and then two specific examples will follow in the next section. Also, the following chapter focuses solely on Navigation, so that discussion will be deferred until then. Figure 13 is the GNC SIMULINK model.

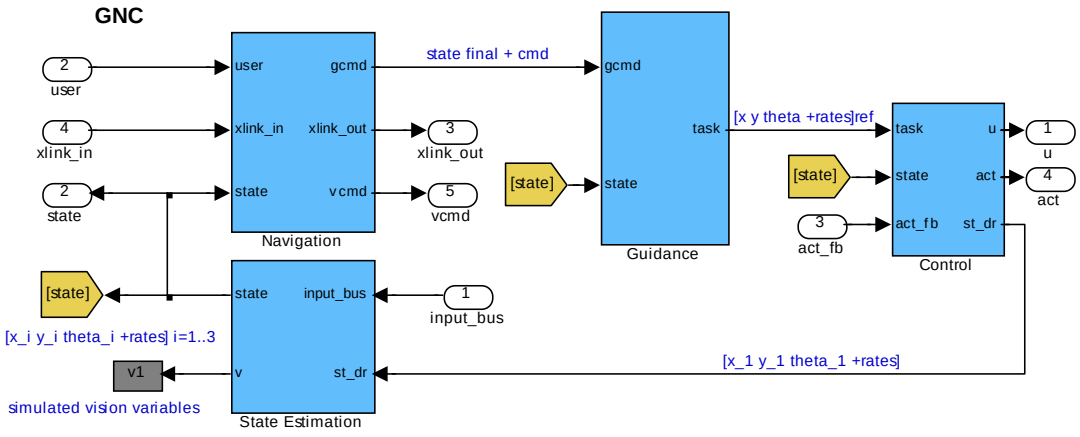


Figure 13. GNC SIMULINK model

The current system state estimation, the user's final desired state, and a command from the onboard autonomy module are input to the guidance module so it may manage the guidance mode, or the manner in how the guidance system behaves. In general, the system state consists of the coordinates of each robot, the attitude of each robot, and the rates at which each of those are changing. Other algorithms may use other parameters as part of the state, such as accelerations, or the positions of control devices. Obviously, these types of changes will require adjustments to several areas of the system.

The guidance module has to work towards the timely movement of the robots to the desired end state while also avoiding collision with other robots and the floor barriers. Two basic methods are here proposed. First, the guidance module can act as a trajectory planner: it will take into account the current and desired states, and then calculate all the control inputs to be applied over time to move the system to the desired end state. Second, the guidance module could consider the current and desired states, and then calculate a task for the controller that is valid for only that instant. Both methods have their advantages and disadvantages; the designer must evaluate the following trade space to determine which type of guidance mode to develop.

Simplicity and computation considerations: Although not necessarily true in every case, trajectory planning guidance systems tend to be much more complicated and computation intensive, while instantaneous tasking systems tend to have very light computation requirements. This fact is rather intuitive considering that trajectory planning algorithms have to calculate many points along a path over time vice a single point in time. Even if a complete trajectory can be calculated in a matter of seconds, it can prevent real-time trajectory updates. Instantaneous tasking algorithms, conversely, only use current information to quickly determine what needs to be accomplished at that particular instant, and can therefore be useful on real-time systems.

Deterministic and optimality considerations: There is no comparison between trajectory planning algorithms and instantaneous tasking algorithms when considering optimality. Trajectory planning systems can calculate and (attempt to) minimize cost functions, giving at the very least a performance index relative to other trajectories. Instantaneous tasking algorithms do not take into account the entire path, and therefore

cannot predict the cost to complete it. In the space environment, using non-optimized algorithms with limited fuel is not recommended, however they provide something to compare against in the AMPHIS lab, and can work with slower processors.

One example of each type of guidance mode is provided in later sections: the Direct Calculus of Variation method is a trajectory planning algorithm and the Artificial Potential Function method is an instantaneous tasking algorithm. Many other guidance and control algorithms could be developed. For example, some programs can provide theoretical optimal solutions to similar problems and could be used to find a performance benchmark. (Ref. [18])

Finally, the control module takes the current state, the task given to it by the guidance system, and feedback provided by the actuators to determine the control inputs to the actuators. A popular controller used in the base configuration is the PID controller. The control module must also provide other functions. Once the PID controller has determined the necessary accelerations needed from the system dynamics, a control mapping must decide how all the available control devices will contribute to the control effort. Each of these signals must then be translated into actuator commands. An integrator, or system plant is also required to model the kinematics of the system; this is required for simulation, but can is as a dead reckoning solution for state estimation. Reference [6] contains an in depth discussion on AMPHIS control.

G. WINDOWS XP COMPUTER SOFTWARE DESIGN

The Windows XP computer has two basic functions: first it acts as a conduit for Wireless LAN communications. This intermediate platform is necessary because there are no wireless adapters available for the xPC Target computer. Therefore, all incoming and outgoing communications must be accomplished with the Windows XP computer, and relayed to/from the xPC Target computer. The second function of the Windows XP computer is to provide artificial vision processing and control. These functions are accomplished here for two reasons: 1) to help distribute the computational load across processors; and 2) because the MATLAB functions that control the LIDAR are not compatible with xPC Target.

The resulting software architecture on the Windows XP computer is relatively simple compared to the xPC Target computer. One module handles all of the LIDAR control and processing, while the second acts an External Connection module similar to the one on the xPC Target computer. Operation of the LIDAR is discussed fully in the Navigation Chapter. Figure 14 is the top level architecture of the Windows XP computer. The code that controls the LIDAR is included in the Appendix.

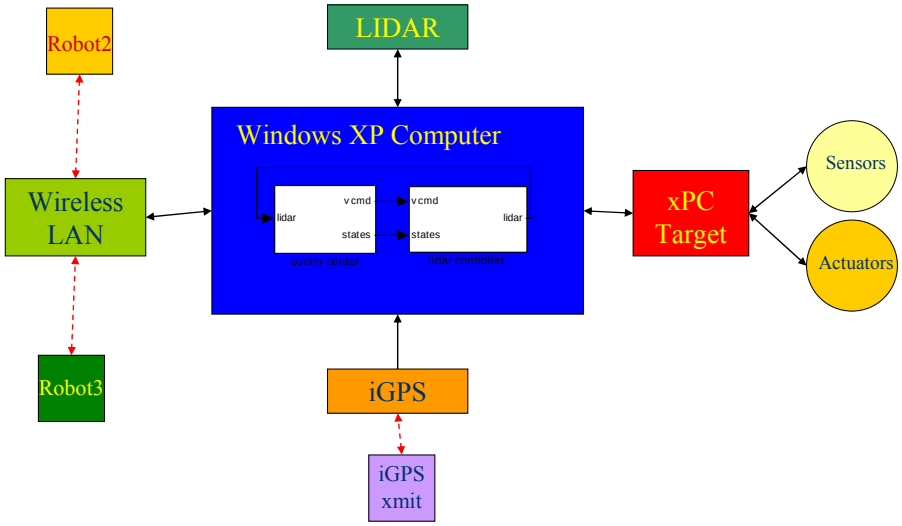


Figure 14. Top level architecture of the Windows XP computer

It is important to restate here another important function the Windows computer may provide. The Indoor GPS system requires a proprietary program named "Work Space" to interpret the signals received by the onboard iGPS receiver. This program runs only on Windows; therefore, either the onboard Windows computer can connect directly to the iGPS receiver with a serial cable, or, if a wireless serial relay is available, Work Space can run on an off board Windows computer and communicate with the iGPS receiver via a wireless serial link.

A major complication with the Work Space program is that its protocol has only been developed in C++ (on a LINUX computer). In order to obtain real time information from the Work Space program, the LINUX computer queries the Work Space program via the TCP/IP protocols, processes the information, and then relays the relevant data to the xPC Target machine via the Windows XP computer on the Wireless LAN.

It is also highly desirable to avoid running the Work Space program on the onboard computer. It does not run well concurrently with SIMULINK; in order for the Work Space program answer queries from the off board LINUX computer in real time, it must be run as "Above Normal" priority. Setting the Work Space task to this priority creates an unstable environment which induces the entire Windows computer to freeze or crash unpredictably. Unfortunately, there is no support for the Indoor GPS system, as the company has changed ownership. There are, however, promises that a new format will correct all of these problems in the summer of 2007.

THIS PAGE INTENTIONALLY LEFT BLANK

III. GUIDANCE AND CONTROL EXAMPLES

A. DIRECT CALCULUS OF VARIATION METHOD

One example of a fully developed simulation model was completed using the Direct Calculus of Variation method. In this method, the three position variables for each robot were approximated to vary as high order polynomials. Using polynomials, velocity, acceleration, and jerk can be found through simple theoretical differentiation. The inverse dynamics then directly indicate the control profiles necessary for to achieve the desired position trajectories. Using MATLAB's fminsearch function, the family of polynomials can then be searched for pseudo-optimal results. (Ref. [14], [15], [16])

The following discussion explains the solution to a rest to rest maneuver in detail. Some of the notation is slightly different than the notation used in other sections of this paper to be consistent with Reference [17]. The configuration in this simulation differs from the base configuration mentioned earlier. The major differences are:

1) The thruster type: instead of using dual fore/aft thrusters, a single free-rotating variable vectored thruster is modeled. It is assumed to always create thrust through the robot's center of mass.

2) The artificial vision sensor: a digital camera which takes two dimensional images and a vision processing computer is responsible for determining were the other robots are relative to the robot on which the camera is mounted. A camera control algorithm controls the camera based on the predicted location of the target robot. The camera will turn to the desired bearing, take a photograph, and pass it to the state estimation module. The camera will alternate photographs between multiple robots. Figure 15 displays the Finite State Machine that will control the camera.

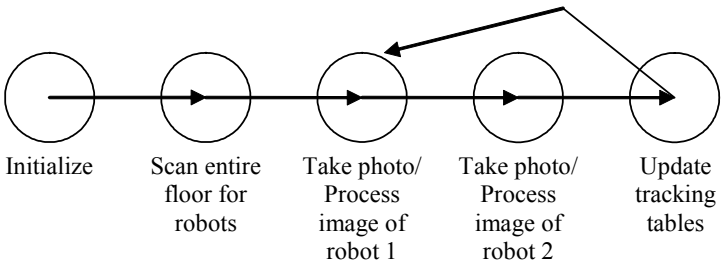


Figure 15. Finite state machine for camera control

A diagram of this setup is in Figure 16. Without loss of generality, only normalized control forces were calculated.

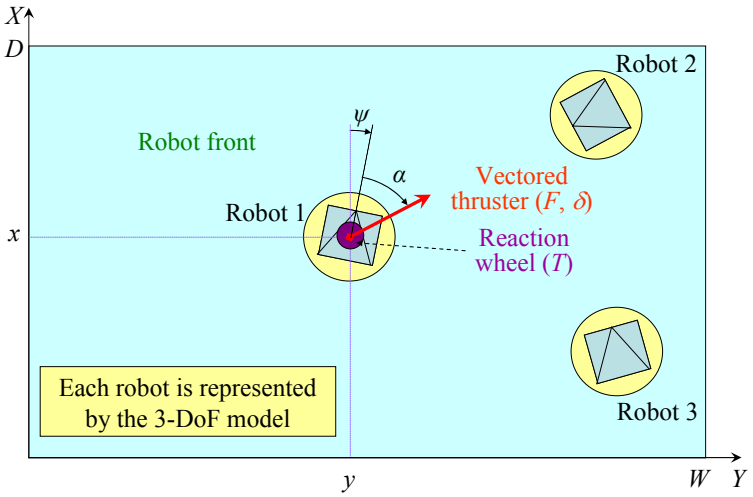


Figure 16. Diagram of the model used for the Direct Calculus of Variation method (Ref. [17])

The problem is first defined mathematically. The system of nonlinear equations driving each robot's dynamics ($i = \overline{1,3}$) is given below:

$$\dot{x}^i = u^i$$
$$\dot{y}^i = v^i \qquad \dot{\psi}^i = \omega^i$$
$$\dot{u}^i = F^i \cos(\psi^i + \alpha^i) \qquad \dot{\omega}^i = T^i$$
$$\dot{v}^i = F^i \sin(\psi^i + \alpha^i) \qquad \dot{\alpha}^i = \delta^i$$

The seven states per robot are its x and y coordinates, x and y, components of its linear velocity, u and v, respectively, the attitude angle ψ (defining robot's orientation with respect to the x-axis), the angular velocity ω controlled by the reaction wheel, and the angle α defining the direction of thrust with respect to robot front. Three available

controls (per robot) are the magnitude of its linear acceleration $F^i = \frac{Thrust^i}{m^i}$ ($0 \le F^i \le F^i_{\max}$), the control input δ affecting orientation of the thrust and the angular acceleration $T^i = \frac{Torque^i}{I^i}$ ($\left|T^i\right| \le T^i_{\max}$). (Ref. [17])

While maneuvering, all robots ($i = \overline{1,3}$) must obey the geometrical constraints of the arena: $0.5MSD \le x^i(t) \le D - 0.5MSD$, $0.5MSD \le y^i(t) \le W - 0.5MSD$, $t \in \left[t_0, t_f\right]$ (where MSD stands for minimum safe distance between two robots and is equal to the diameter of the circles drawn on Figure 16 around each robot), and avoid collisions with other robots:

$\left(x^i(t) - x^k(t)\right)^2 + \left(y^i(t) - y^k(t)\right)^2 - MSD^2 \ge 0$, $\forall\ i,k = \overline{1,3},\ i \ne k$, $t \in \left[t_0, t_f\right]$. (Ref. [17])

It is required to satisfy the following sets of boundary conditions per each robot ($i = \overline{1,3}$):

$$\begin{array}{llll} x^i(t_0) = x^i_0 & x^i(t^i_f) = x^i_f & \dot{x}^i(t_0) = u^i_0 & \dot{x}^i(t^i_f) = u^i_f \\ y^i(t_0) = y^i_0 & y^i(t^i_f) = y^i_f & \dot{y}^i(t_0) = v^i_0 & \dot{y}^i(t^i_f) = v^i_f \\ \psi^i(t_0) = \psi^i_0 & \psi^i(t^i_f) = \psi^i_f & \dot{\psi}^i(t_0) = \omega^i_0 & \dot{\psi}^i(t^i_f) = \omega^i_f \end{array}$$

$$\begin{array}{ll} \ddot{x}^i(t_0) = F^i_0 \cos(\psi^i_0 + \alpha^i_0) & \ddot{x}^i(t^i_f) = F^i_f \cos(\psi^i_f + \alpha^i_f) \\ \ddot{y}^i(t_0) = F^i_0 \sin(\psi^i_0 + \alpha^i_0) & \ddot{y}^i(t^i_f) = F^i_f \sin(\psi^i_f + \alpha^i_f) \\ \ddot{\psi}^i(t_0) = T^i_0 & \ddot{\psi}^i(t^i_f) = T^i_f \end{array} \quad \text{(Ref. [17])}$$

In general, the performance index includes three appropriately weighted terms. The first one, t^1_f, assures minimum transition time for the first robot, the second one, $\left|t^2_f - t^1_f - \Delta_t\right| + \left|t^3_f - t^2_f - \Delta_t\right|$, guarantees sequential (Δ_t-second apart) joining the final formation, and the third one, $\sum_{r=1}^{3} \int_{t_0}^{t^i_f} F^i dt$, takes care of minimizing overall gas consumption to produce thrust. (Ref. [17])

To generate quasi-optimal collision-free trajectories for all three robots in real time (and to be able to update them every 2-3 seconds) the direct method of calculus of variations was chosen. (Ref. [14]) To apply it we need to introduce an independent argument τ^i for each robot ($i = \overline{1,3}$) and using the corresponding speed factors λ^i (different for each robot) rewrite the original system as

$$
\begin{aligned}
x'^i &= u^i/\lambda^i & & \\
y'^i &= v^i/\lambda^i & \psi^{i\prime} &= \omega^i/\lambda^i \\
u'^i &= F^i\cos(\psi^i+\alpha^i)/\lambda^i & \omega^{i\prime} &= T^i/\lambda^i \\
v'^i &= F^i\sin(\psi^i+\alpha^i)/\lambda^i & \alpha^{i\prime} &= \delta^i/\lambda^i
\end{aligned}
\quad \text{(Ref. [17])}
$$

Next, three reference functions (per robot) are established for coordinates x^i and y^i, as well as for the attitude angle ψ^i: $P_x^i(\tau^i)$, $P_y^i(\tau^i)$ and $P_\psi^i(\tau^i)$, respectively ($i=\overline{1,3}$). If polynomials are used, then the order of polynomial to use is defined by the number of boundary conditions, which in this case the minimum order of approximating polynomials is five. (Ref. [14]) For this specific problem to have an additional flexibility (to allow avoiding collisions), the order of polynomials was increased by two to be able to vary the third derivative of x^i, y^i and ψ^i, $i=\overline{1,3}$ at both ends. (Ref. [17])

Explanation of the optimization routine follows. Given the boundary conditions, nine reference polynomials, $P_x^i(\tau^i)$, $P_y^i(\tau^i)$ and $P_\psi^i(\tau^i)$, $i=\overline{1,3}$, have to be determined, and their coefficients computed using the boundary conditions and initial guesses on the third derivatives $x_0^{i\prime\prime\prime}$, $x_f^{i\prime\prime\prime}$, $y_0^{i\prime\prime\prime}$, $y_f^{i\prime\prime\prime}$, $\psi_0^{i\prime\prime\prime}$, $\psi_f^{i\prime\prime\prime}$, $i=\overline{1,3}$. These variables along with the lengths of three virtual arcs τ_f^i form the vector of variable parameters Ξ. Next, applying inverse dynamics, the remaining states can be solved for numerically. Specifically, start by dividing each virtual arc τ_f^i ($i=\overline{1,3}$) onto *N*–1 equal pieces $\Delta\tau^i=\dfrac{\tau_f^i}{N-1}$ so that there are *N* equidistant nodes $j=\overline{1,N}$ along each virtual arc. For each robot, all states at the first point $j=1$ (corresponding to $\tau_1^i=\tau_0^i=0$) are defined. Additionally, define $\lambda_1^i=1$, $i=\overline{1,3}$. (Ref. [17])

For each of the subsequent *N*–1 nodes $j=\overline{2,N}$, the current values of robots' coordinates and attitudes are calculated using each corresponding polynomial: $x_j^i=P_x^i(\tau_j^i)$, $y_j^i=P_y^i(\tau_j^i)$ and $\psi_j^i=P_\psi^i(\tau_j^i)$, $i=\overline{1,3}$. Then, using the inverse dynamics for the first four equations of the system, the sum of angles ψ_j^i and α_j^i, and current control acceleration are calculated: $\psi_j^i+\alpha_j^i=\arctan\left(\dfrac{y_j^{i\prime\prime}}{x_j^{i\prime\prime}}\right)$, $F_j^i=\lambda_j^i\sqrt{x_j^{i\prime\prime 2}+y_j^{i\prime\prime 2}}$. (Ref. [17])

Inverting the last equation of the system and using the first of two equations, the second control is obtained: $\delta_j^i = \lambda_j^i \alpha_j^{i\prime} = \lambda_j^i \left(\frac{x_j^{i\prime\prime\prime} y_j^{i\prime\prime} - x_j^{\prime\prime i} y_j^{i\prime\prime\prime}}{y_j^{i\prime\prime}} \cos^2 \psi_j^i - \psi_j^{i\prime} \right)$. (Ref. [17])

From the first two equations of the system, the current speed, $V_j^i = \sqrt{u_j^{i2} + v_j^{i2}}$, where $u_j^i = \lambda_j^i x_j^{i\prime}$, $v_j^i = \lambda_j^i y_j^{i\prime}$, is defined, and therefore, the elapsed time for each robot can be determined: $\Delta t_{j-1}^i = 2 \frac{\sqrt{\left(x_j^i - x_{j-1}^i\right)^2 + \left(y_j^i - y_{j-1}^i\right)^2}}{V_j^i + V_{j-1}^i}$. (Ref. [17])

Now, the current values of the speed factor are given by $\lambda_j^i = \frac{\Delta \tau^i}{\Delta t_{j-1}^i}$, and the current time for each robot is defined as $t_j^i = t_{j-1}^i + \Delta t_{j-1}^i \quad (t_1^i = 0)$. (Ref. [17])

Finally, the equations are inversed for the robots' attitude to get the third control $\omega_j^i = 2 \frac{\psi_j^i - \psi_{j-1}^i}{\Delta t_{j-1}^i} - \omega_{j-1}^i$ and $T_j^i = \frac{\omega_j^i - \omega_{j-1}^i}{\Delta t_{j-1}^i}$. (Ref. [17])

Once all states along the trajectories are computed, the performance index is found. Employing the vector of weighting coefficients **w** ($\sum_{h=1}^{3} w_h = 1$), $J = w_1 t_f^1 + w_2 \left(\left| t_f^2 - t_f^1 - \Delta_t \right| + \left| t_f^3 - t_f^2 - \Delta_t \right| \right) + w_3 \sum_{r=1}^{3} \sum_{j=0}^{N-1} F^i \Delta t_j^r$ and form the aggregate penalty using an appropriate four-component vector of weighting coefficients **k** ($\sum_{q=1}^{4} k_q = 1$):

$$\Delta = \left[k_1, k_2, k_3, k_4, k_4, k_4 \right] \begin{bmatrix} \sum_{i=1}^{3} \left(\max_j \left(0; F_j^i - F_{\max}^i \right) \right)^2 \\ \sum_{i=1}^{3} \left(\max_j \left(0; \left| T_j^i \right| - T_{\max}^i \right) \right)^2 \\ \sum_{i=1}^{3} \left(\left(\max_j \left(0; \left| x_j^i - \frac{D}{2} \right| - \frac{D}{2} + MSD \right) \right)^2 + \left(\max_j \left(0; \left| y_j^i - \frac{W}{2} \right| - \frac{W}{2} + MSD \right) \right)^2 \right) \\ \max_j \left(0; MSD^2 - \left(x^1(t_j^*) - x^2(t_j^*) \right)^2 - \left(y^1(t_j^*) - y^2(t_j^*) \right)^2 \right), \ * = \arg\min_{i=1,2} (t_f^i) \\ \max_j \left(0; MSD^2 - \left(x^1(t_j^*) - x^3(t_j^*) \right)^2 - \left(y^1(t_j^*) - y^3(t_j^*) \right)^2 \right), \ * = \arg\min_{i=1,3} (t_f^i) \\ \max_j \left(0; MSD^2 - \left(x^2(t_j^*) - x^3(t_j^*) \right)^2 - \left(y^2(t_j^*) - y^3(t_j^*) \right)^2 \right), \ * = \arg\min_{i=2,3} (t_f^i) \end{bmatrix}$$

(Ref. [17]) Note that the last three terms in the compound penalty are quite tricky because robots' coordinates have to be interpolated so that they correspond to the same instants of time.

Finally, a standard nonlinear constrained minimization routine is used to minimize the performance index while keeping the penalty within the certain tolerance: $\min_{\underline{\Xi}} J\Big|_{\Delta\leq\varepsilon}$. (Ref. [17])

A rest to rest maneuver was simulated from an arbitrary starting position to a close-in, triangular final position. Four frames from the bird's eye view animation are provided in Figure 17. Robot 1 (bottom left), Robot 2 (top center), and Robot 3 (right center) perform the rest to rest maneuver in approximately 45 seconds. Frame (a) depicts the starting position, frame (b) and frame (c) depict intermediate positions, and frame (d) depicts the final position with the ground tracks to achieve that position. The front side of each robot is indicated buy a line extending from its center. The direction and magnitude of the rotating thruster is also indicated by the plume extending from the robots. After defining the initial absolute position and final relative position, the algorithm varied the final absolute position, time, and time factor step size ($\Delta\tau$), initial and final jerk (the derivative of acceleration) to achieve this final position without collision and in timely manner. The guidance algorithm achieved these results by computing polynomials for x, y, and ψ for each of the robots such that all positions, velocities, and accelerations met the given boundary conditions. The control profiles were calculated from the inverse dynamics. (Ref. [17])

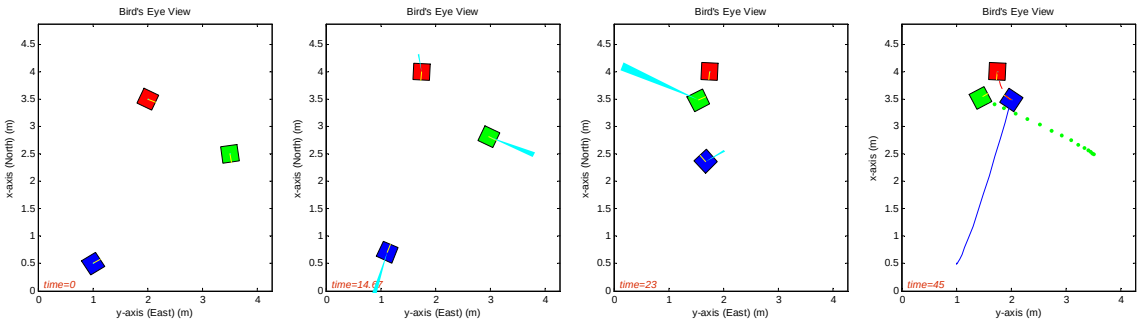


Figure 17. Example sequence at 0 (a), 15 (b), 23 (c) and 45 seconds (d) (Ref. [17])

A summary of all parameters are shown as functions of the time factor τ in Figure 18.

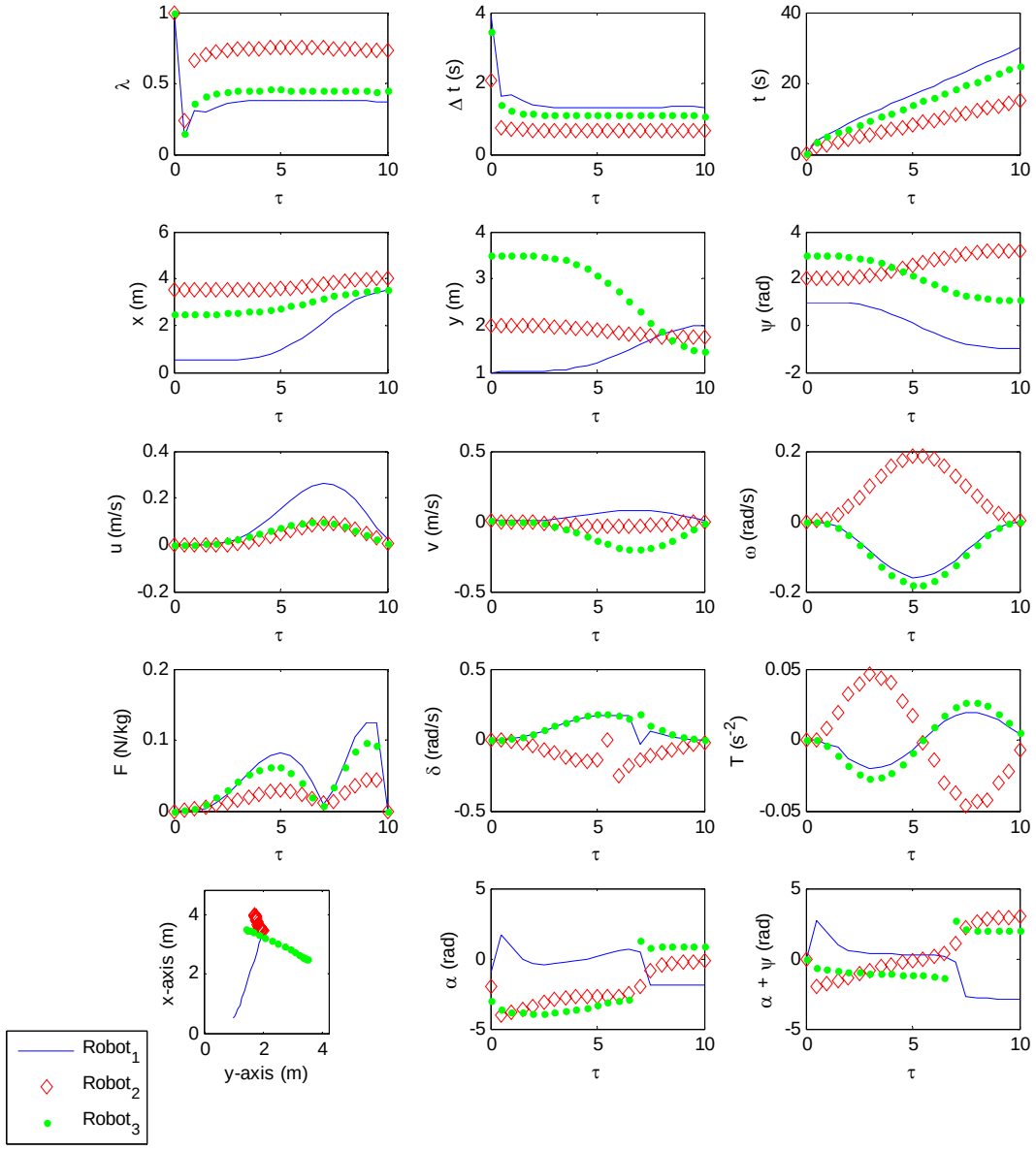


Figure 18. Summary of parameters for Direct Calculus of Variation method (Ref. [17])

The next step in this implementing this algorithm would be to optimize it to point of being able to provide real time, closed loop solutions.

B. ARTIFICIAL POTENTIAL FUNCTION GUIDANCE

In contrast to the Direct Calculus of Variation Method, Artificial Potential Function Guidance (APFG) provides a quick way to calculate a control input. In general, APFG can be implemented in various ways, including as a trajectory planner. Conceptually, APFG is generally explained as positive and negative potential fields which the weighted sum provides gradient to follow. This gradient hopefully ends in at the final desired state, or global minimum of the potential function.

The algorithm developed here takes some of the ideas from APFG and facilitates easy implementation with a PID controller. The robot using this algorithm looks for obstacles (i.e., other robots) that could potentially be in its way to its final destination, or in its "collision zone." It also checks if any obstacles are too close to it, or in its "safety zone." If either of these two cases exists, a correction, or, avoidance vector is added to the vector which directs the robot to its final desired destination. Figure 19 illustrates this concept.

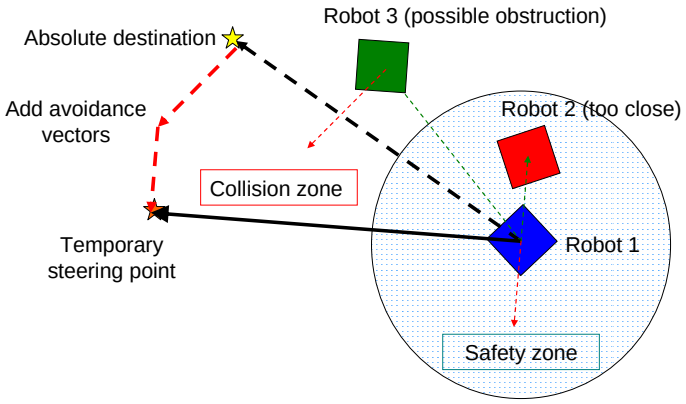


Figure 19. APFG concept

In the case that there is a robot in the collision zone, a vector tangent to the direction of the obstacle is added to the final destination vector. This correction would allow the robot to circle around an obstacle until the path to its desired location is clear. In the case that there is an obstacle that is within the safety zone of the robot, a correction vector in the opposite direction is added to the final point. The resulting steering point is

therefore the vector sum of up to five vectors: the unobstructed final destination; up to two tangent vectors if there are obstacles between the robot and the final destination; and up to two repulsive vectors, if there are obstacles too close to the robot. This combination of vectors provides a temporary point for the PID controller to steer towards. As the robots move and the system changes, so does the steer point. Once the path to the final destination point is clear the robot can proceed directly to it.

APFG offers a quick solution, but suffers from being non-deterministic, and non-optimal. There is also a problem with local minima. This case is analogous to several corrective vectors being symmetric and actually canceling each other out, so no corrective vector is applied and a collision could result. One way to help avoid this situation is by weighting the vectors differently. For example, weighting a repulsive vector by 1/d, where d is the distance between the robot and the obstacle, will give a stronger repulsion as d decreases. Multiplying the vector by the velocity will decrease the repulsion as the robot moves slower, as for the case with docking. Furthermore, weighting the vector by other functions, such as Ae^{-d} where A is a constant, or user defined gain, will give the system even different behavior.

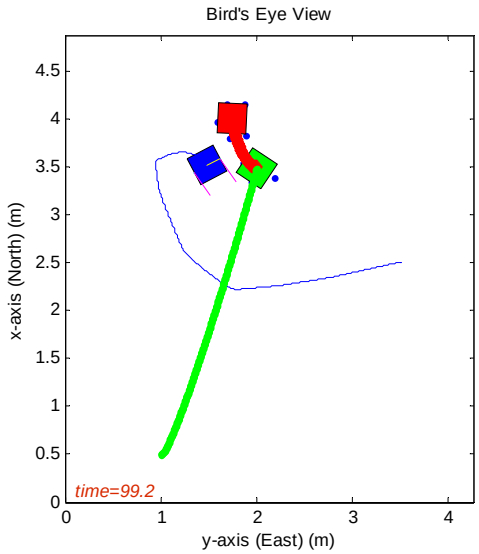


Figure 20. APFG simulation output

A simulation was conducted using an APFG algorithm with the base configuration described in the previous chapter. The two other robots followed the trajectories calculated in the previous example; the primary robot was left to navigate on its own to the absolute final position without colliding with the other robots. The outcome can be seen above in Figure 20, and the parameters versus time are provided in Figure 21.

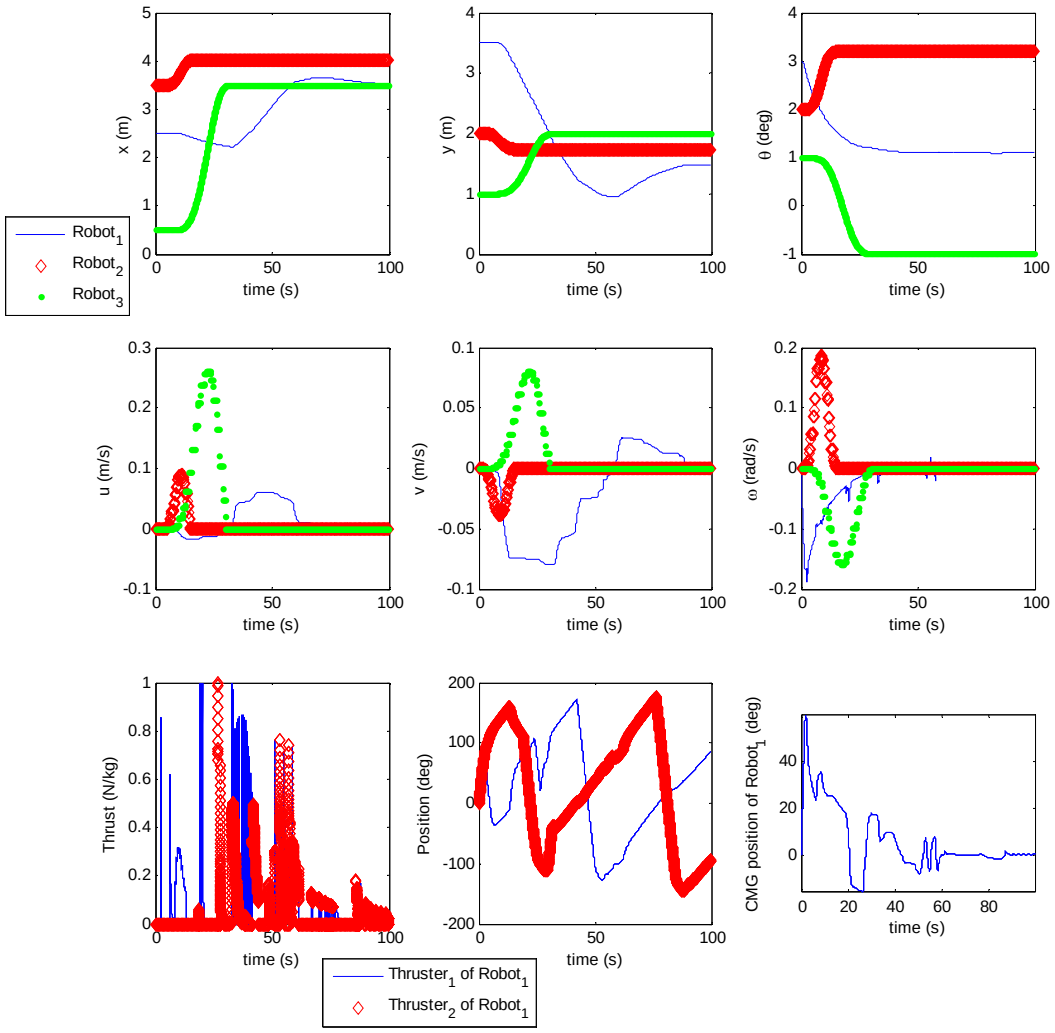


Figure 21. Parameters vs. time for a APFG simulation

Code written for this APFG is included in the Appendix. Information on the control system for this configuration is detailed in Reference [6].

IV. NAVIGATION

There are two main purposes of Navigation: state estimation and onboard autonomy. State estimation is the process of evaluating all of the system data and making the best possible estimates of positions, attitudes, and rates for all of the robots on the floor. Generally two sets of sensors are needed for the AMPHIS testbed: onboard sensors for each robot to determine where they themselves are, and at least one other sensor to determine where the other robots are, such as LIDAR.

A. STATE ESTIMATION

Valid knowledge of the system state is required to effectively navigate from point A to point B. In case of the described base configuration, the state consists of eighteen variables: the coordinates and attitudes of all three robots, and each of their rates. The state could be expanded to include the positions of some sensors or actuators, or accelerations, etc. if desired. The state is estimated in two separate parts: the robot estimates its own state, and is estimates the state of the other two robots. The system can easily be configured to uses one of the following sensor groups to determine the state:

Robot determines its own state using:

- iGPS and the gyro
- accelerometers, gyro, and kinematics integrator
- kinematics integrator (simulation or open-loop control)
- lookup table (simulation only)

Robot determines the state of the other robots using:

- data transmitted on the wireless LAN (via UDP)
- LIDAR
- lookup table (simulation only)

The two inputs to the state estimation module are the “input_bus,” which carries all of the data from the onboard sensors and the wireless LAN, and the “st_dr,” which stands for

"state determined by dead reckoning." This is the output from the kinematics integrator. Figure 22 is the state estimation SIMULINK model.

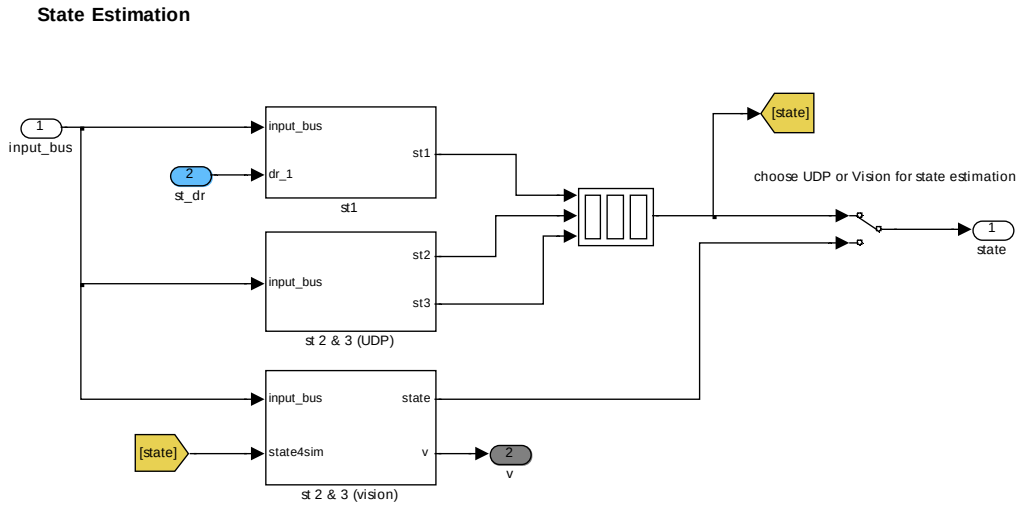


Figure 22. State estimation SIMULINK model

The state variables that are determined by the iGPS or from the kinematics integrator are explained it Reference [6]. The state variables estimated from the LIDAR are discussed next.

LIDAR gives bearing and range information in a plane circling it. The next section will discuss how pose estimations of the robots on the floor are made from LIDAR date. Once pose estimations are made from the LIDAR data, pose estimations, along with the estimation a robot has made of its own state, is combined to create a entire state. The function "pe2st," or, "pose estimation to state variable," takes the estimated coordinates of itself (the LIDAR is mounted in the center of the robot), along with the relative bearings, ranges and orientations of the other robots, and creates a three by three position matrix from it in absolute coordinates. The columns of this matrix represent robot 1, robot 2, and robot 3, respectively. The rows represent the x coordinate, y coordinate, and attitude angle, respectively. As can be seen from Figure 23, the remaining state variable (the rates) are found using the derivative block and are then

combined to them with the vertical concatenation block. The Kalman filter will significantly upgrade these values once employed.

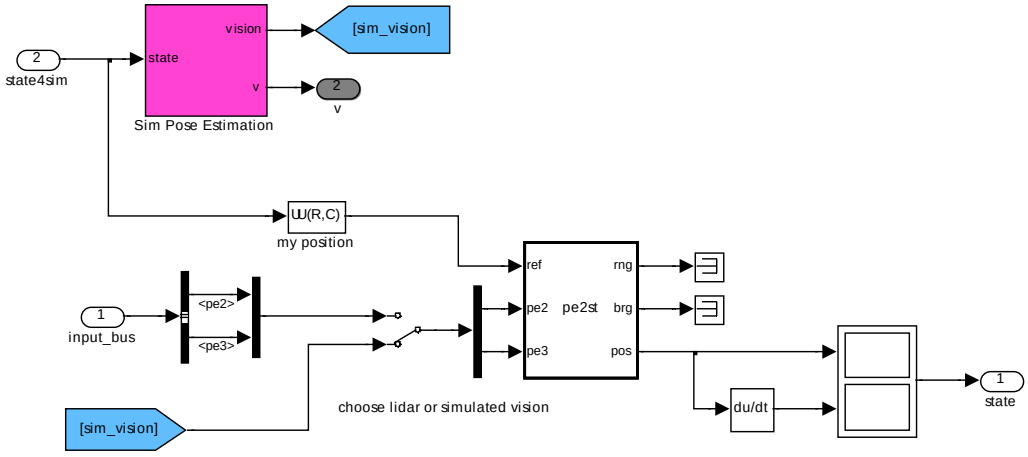


Figure 23. State estimation from vision module

For simulation and testing, a pose estimator simulator lets the accuracy and update rates be defined for pose estimation. Figure 24 is a model of the pose estimator simulator.

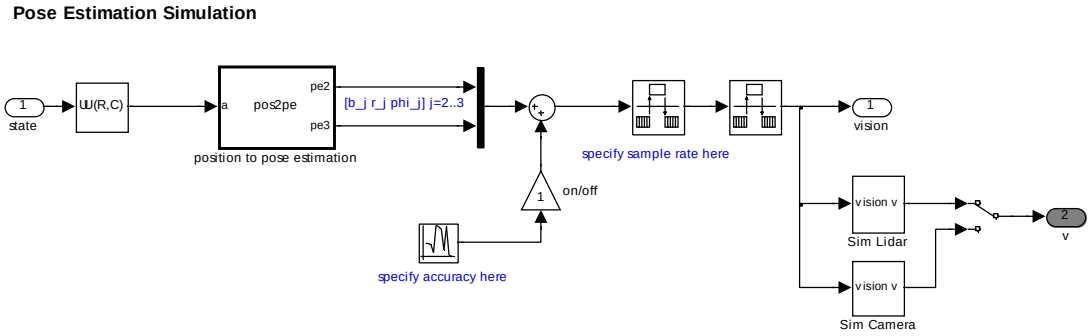


Figure 24. Pose estimation simulator

B. POSE ESTIMATION STRATEGIES USING ARTIFICIAL VISION

Pose estimation is the process of determining an object's position and orientation relative to the artificial vision sensor's position and orientation. Several variable sets can be used to describe this information. In general, the number of variables needed to describe a pose is equal to the degrees of freedom of the system. There are several

commonly available artificial vision sensors for that can be used for pose estimation. Pose estimation using LIDAR and digital imaging cameras are discussed here. Range (or stereo) cameras which deliver three dimensional information and omni-directional cameras (Bearing/Range) are two other viable sensors for this application.

1. LIDAR (Bearing/Range)

LIDAR gives bearing and range information in a single plane. Since the LIDAR is mounted level at the top center of the robot, the plane that it measures is parallel to the floor at an equidistant height. It has been experimentally shown that reflective surfaces and especially the special reflective LIDAR tape can greatly improve the read. The LIDAR starts at a designated point on the physical unit and returns ranges at every step angle moving clockwise. For the base configuration, the first angle is at 0˚ (the flat part of the LIDAR case) and the step angle is equal to 0.625˚. One full revolution therefore provides 575 range measurements (Figure 25).

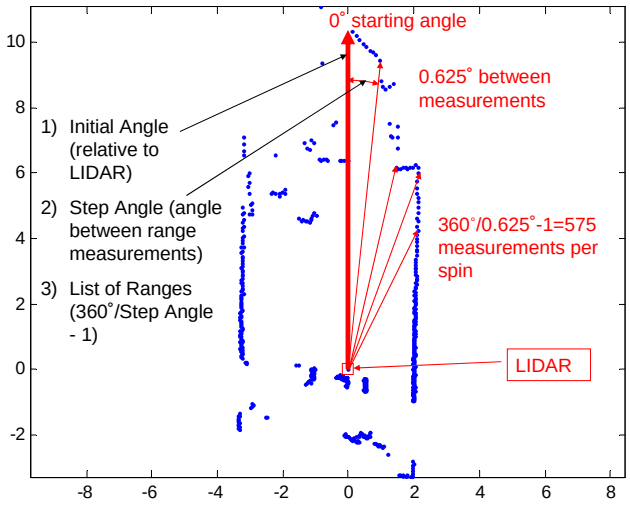


Figure 25. LIDAR operation and basic data return

There are two problems that must be solved to use a LIDAR effectively with the AMPHIS test bed. Firstly, other robots must be distinguished from the rest of the objects in the room seen by the LIDAR. Secondly, the pose estimation of those robots must be made. The first problem is somewhat artificial, since in space, there would not be any clutter in range of the proximity operations. Therefore, it is desired to spend little

resources solving this problem. The easiest way to determine what points correspond to robots and which do not is by finding the floor. If there are points on the floor, they are assumed to be robots.

The first step to finding is to convert the bearing and range information into x_L and y_L coordinates. These coordinates are relative to the LIDAR, not the floor. This transformation is straightforward using polar to Cartesian transformations (Figure 26).

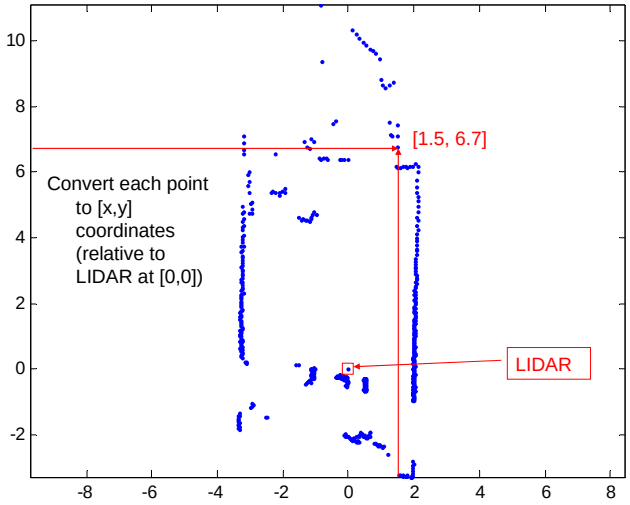


Figure 26. Convert LIDAR data to Cartesian coordinates

Taking into account the robot's position and attitude on the floor, a transformation can then me made to the points from the LIDAR. If all of these points are rotated by the opposite of its attitude angle, and then shifted in the x and y directions by the amount of its absolute coordinates, the LIDAR data will be shifted into the absolute "floor" coordinate system. The origin of this coordinate system is in the bottom left corner of the floor (represented by the dotted green box) in Figure 27.

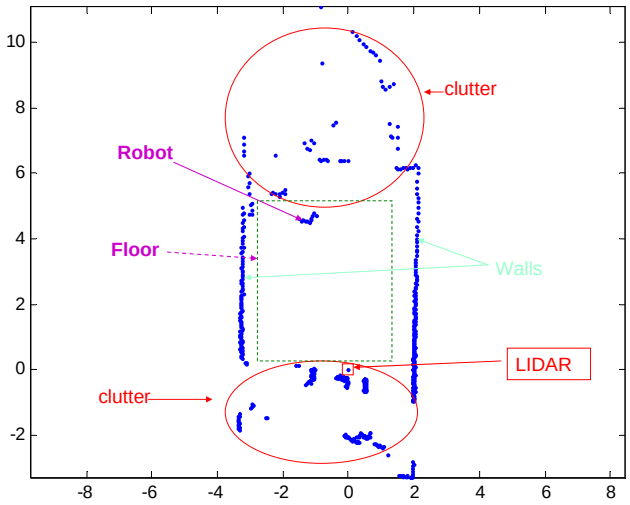


Figure 27. Finding the floor using LIDAR

Objects are then created from the point ranges from the LIDAR. If consecutive points are different by more than a prescribed amount, it is assumed that they belong to different physical objects. Assigning each point to an object make it easier to process the data. Objects that are too big, too small, or too far away can easily be discarded (Figure 28). It is important to note that there is a discontinuity at the starting point (0˚/360˚). Therefore, those points should be considered together as a possible single object.

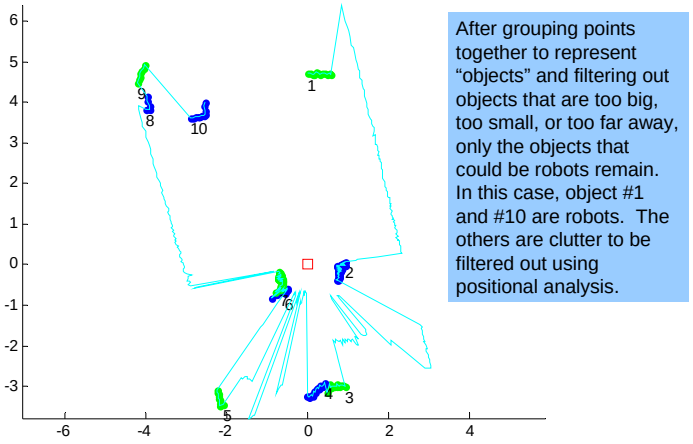


Figure 28. Assigning points to objects

Once objects are found, numbered, and their absolute positions are known, the objections on the floor are determined (because they are inside the square bounded by (0,0) and (14,16) feet), their relative bearings, ranges, and orientations are found (Figure 29). The relative bearing is estimated by finding the median bearing between the extreme ends of an object. The range is found by adding approximately six inches (half the width of a robot) to the minimum range of the points that make up that object. Finally, the orientation can be estimated by using linear regression on the line or lines made from the edges of the LIDAR return.

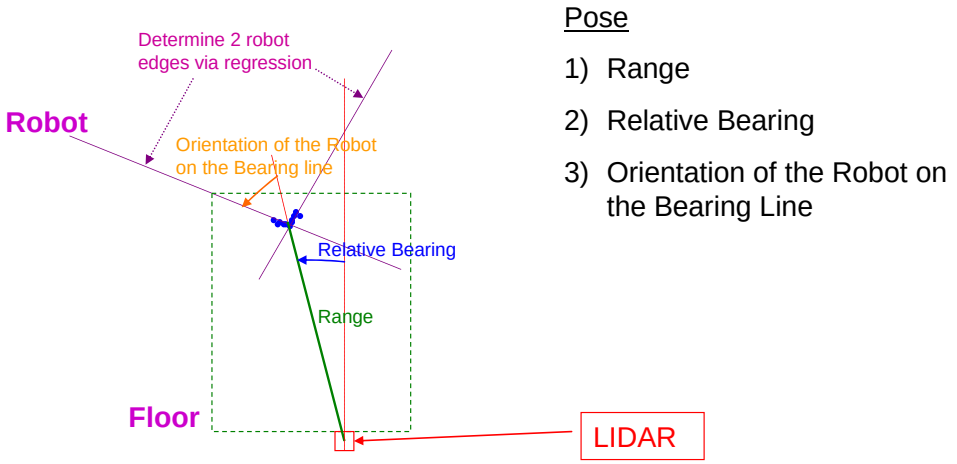


Figure 29. Estimating the pose of LIDAR objects

The LIDAR's ability to track objects was successfully tested using these techniques. The LIDAR scanned the room, assigned each point to an object, filtered out small objects and the transformed the points into the floor's coordinated system. The LIDAR processing algorithm successfully identified objects on the floor as robots, and ignored all other robots. As an object (in this case, a person) moved about the floor, the LIDAR could track and plot the objects position and just under 0.5 Hz (once every 2.2 seconds) on the PC104 onboard computer. Previous experiments on faster computers that were conducted without plotting the results real-time were able to reach update rates of over 1 HZ (less than a second per update).

2. Camera (2D Photograph)

There are several ways to determine a pose from a two dimensional image (from camera, for instance), but only two methods will focused on here. The pose

determination algorithms discussed here are fairly straightforward once certain key features, such as corners or edges, are ascertained from the photograph. But obtaining these key features autonomously via image processing makes pose determination from a single photo a very complicated problem. Issues that require handling include: separating the object you wish to from the background clutter; determine if there is something between the camera and the object (object obscuration); and determining if the photo you are examining contains enough data to even estimate a pose (i.e. a picture does not encompass the entire object). Image processing techniques must be developed to mitigate these problems. Although some of these image processing techniques will be briefly mentioned, the remainder of this section will focus on the algorithms used once the key features have been found.

a. Using Points

The first algorithm discussed here is a general pose estimation method using key points of a known object. Using the "key points" of an object, such as the corners of a square of known size, the pose can be estimated by solving a non-linear system of equations for the reverse transformation of a three dimensional scene onto a two dimensional plane.

For the purpose of describing the pose estimation algorithm, a coordinate system is chosen similar to the one in 9: the camera is centered at the origin of a left-handed orthogonal coordinate system pointing down the positive z axis. Positive x is to the left, and positive y is up. The z axis is limited to non-negative values since negative values would indicate an object is behind the camera, out of its field of view. The next concept to realize is that a photograph is a projection of a three dimensional scene onto a two dimensional plane. This plane is called the focal, or interpretation plane. The coordinate system and interpretation plane is illustrated in Figure 30.

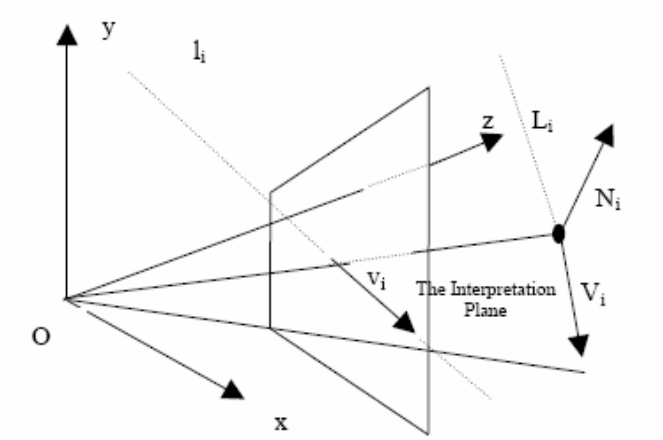


Figure 30. Point pose estimation coordinate system and interpretation plane

At this point, is simpler to attempt to solve the reverse problem to pose estimation, which is to rotate and translate an object of known shape onto the coordinate system and determine how it would appear on the projection plane. Since the object is assumed to be known, it can be defined arbitrarily in a similar coordinate system to that described above. For example, a simple square with a side length equal 1 unit can be defined so it is centered on, and lies completely on the xy plane. Each corner is then represented by four column vectors creating the object matrix O:

$$O = \begin{bmatrix} -.5 & -.5 & .5 & .5 \\ -.5 & .5 & .5 & -.5 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

A mathematical expression can represent the rotation and translation of this action. First, the rotation matrix $R_{\alpha\beta\gamma}$ is a direction cosine matrix that will rotate an object about the x, y, and z axes by the amounts α, β, γ respectively. The Euler equation for a left-handed coordinate system is

$$R_{\alpha\beta\gamma} = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}.$$

Next, all of the object's key points will be translated in the x, y and z directions by the amounts a, b, and c respectively. This completes the positioning of the

object in the three dimensional coordinate system, given by the expression $P = R_{\alpha\beta\gamma} O + \begin{bmatrix} a \\ b \\ c \end{bmatrix} \begin{bmatrix} 1 & \dots & 1 \\ 1 & \dots & 1 \\ 1 & \dots & 1 \end{bmatrix}$ where P is the position matrix of the object and the number of columns of ones is equal to the number of points in the object.

The next step in this reverse problem solving track is to project the three dimensional object onto a two dimensional plane. As seen above, the matrix to perform the positioning transformation must be constructed based on how many points make up the object, so there is not a simple formula. Since the focal plane is defined by a constant z=focal length *f*, each point in 3-space is transformed to 2-space by the following formula: [9]

$p = \begin{bmatrix} P_x / P_z \\ P_y / P_z \\ 1 \end{bmatrix} f$. This formula enables the ability to create a "simulated" photograph given a point in 3-space and a focal plane of distance *f*. Again, the simulated photograph would be on the xy plane (the first elements of vector p).

As shown, deriving the formula for transforming points in 3-space to a 2-space projection is straight foreword. Figure 31 illustrates this process.

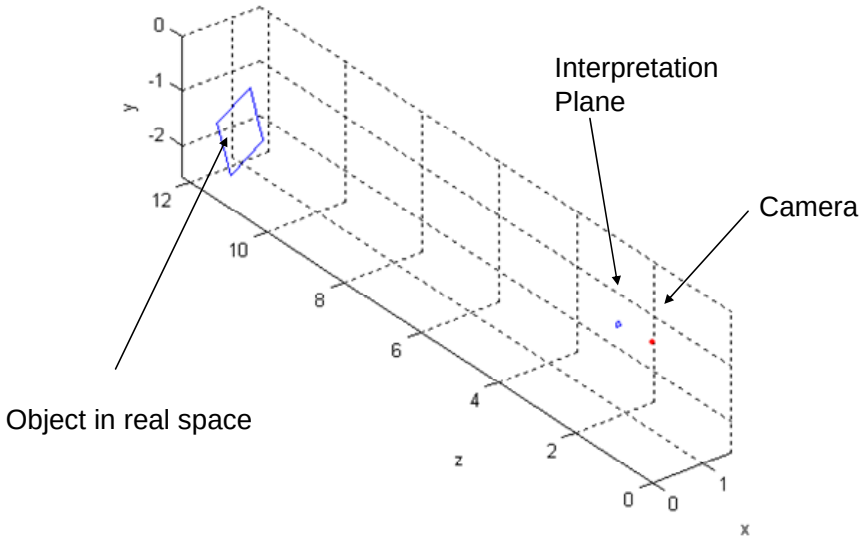


Figure 31. Projection of an object in 3-space to 2-space

Estimating the pose of a known object from a photograph is simply the solving the above problem in reverse. Given a the set of key object points on a photograph, solve for the pose, or the variables $\alpha, \beta, \gamma, a, b, c$. Using symbols, as in MATLAB's symbolic toolbox, the position matrix *P* (defined above) is constructed. Each point in *P* (real space) has a corresponding point in *p* (on the photograph). Each point therefore contributes two equations to our system of equations (one for each of the x and y coordinates of the photograph). From image processing, the x-y coordinates of the photograph are extracted, and provided a solution for our system of equations. Therefore, for N key points on a photograph, a system of 2N nonlinear equations must be solved to yield the pose variables $\alpha, \beta, \gamma, a, b, c$. Figure 32 illustrates this reversed process.

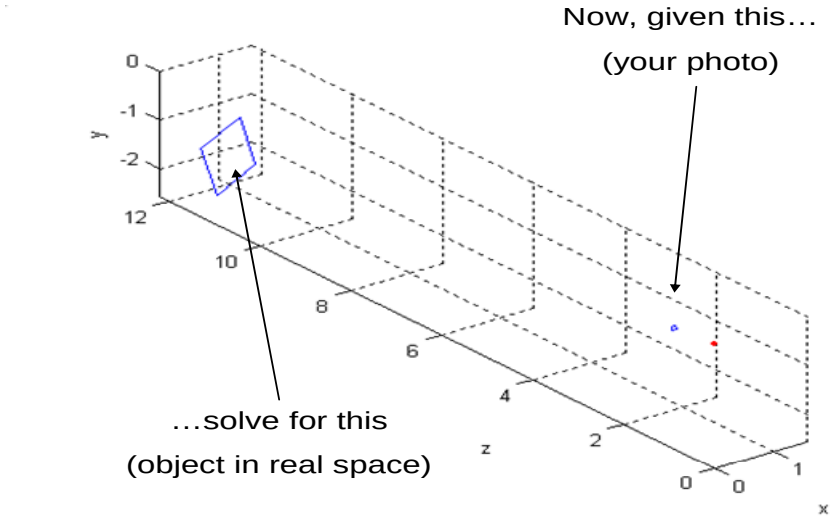


Figure 32. Solving for an object in 3-space from an object in 2-space

This process was implemented in MATLAB. A square was rotated and translated into 3-space. A simulated photograph of that object was taken on the focal plane, shown in blue in Figure 33. Some noise, or error, was then introduced to the points on the two dimensional photo, and MATLAB's *fsolve* function was using to derive the original rotation and translation amounts. A simulated photograph of this derived object was taken on the focal plane, shown in dotted red in Figure 33. These functions are included in the Appendix.

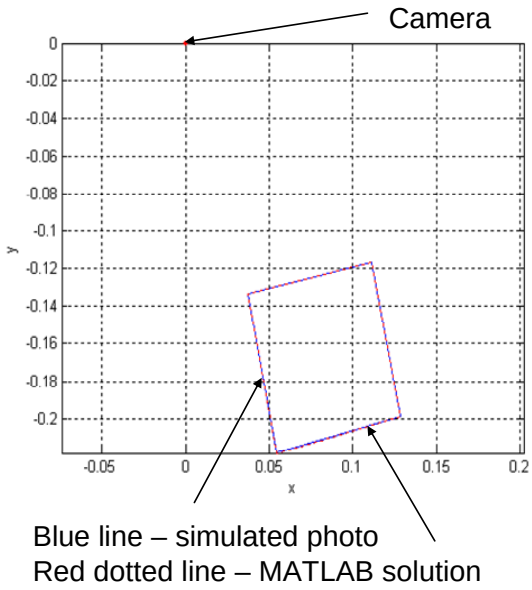


Figure 33. Pose estimation using points demonstration

There are several limitations to the using the pose estimation using points algorithm. One is know as the Necker's cube illusion. A hollow cube has ambiguous poses as can be seen in Figure 34. Due to symmetry, multiple sides can appear to be the nearest to you. Also, a symmetric object will have ambiguous pose solutions.

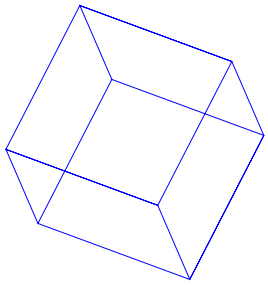


Figure 34. Necker's cube illusion

Another problem that challenges this algorithm is not being able to find all the key points on a photograph. As mentioned, the number of equations required to determine the pose of an object depends on the number of key points in the object. If all the key points cannot be found, a difficult process of trying to limit the object points would ensue.

b. Using Edges

Image processing occurs automatically after a digital photograph is received from the camera. The algorithm will first try to determine how many robots are in view: zero, one, or two. If there are no robots in the field of view, a search routine will have to be conducted. This routine is completed after initialization; once the robots are acquired and tracking has started, the camera will alternate amongst the moving robots and attempt to keep them in its field of view. If two robots are in the photo, it is preferred to center the camera on one robot at a time. If this is not possible, such as the case when one robot is behind the other, accurate pose estimates are very difficult to make.

Each photograph is processed onboard the robot to find and determine the relative positions of the other robot(s). Figures 6a and 6b illustrate an example photograph of one of the robot used in Ref.1 and a simulated photograph assumed to be of Robot 3 as seen by Robot 1. The image processor locates the three vertical support structures from the image determines the robot's relative position from the know size and shape of the robot(s) in the field of view.

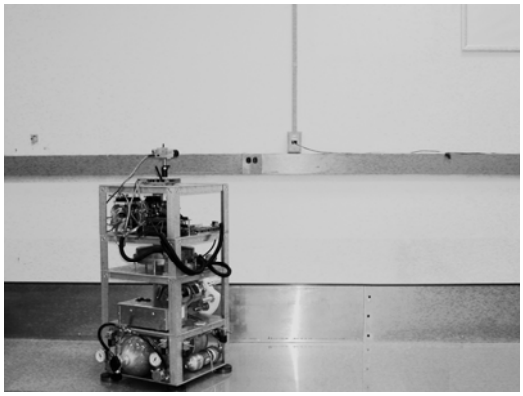


Figure 35. Actual image taken from Robot 1

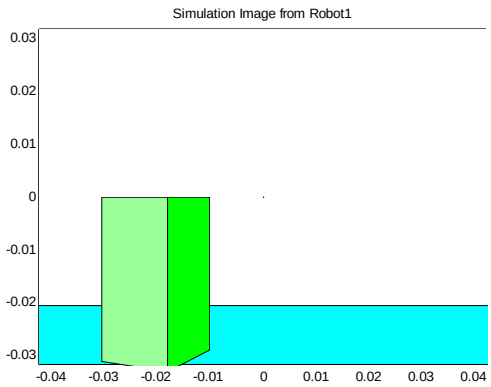


Figure 36. Simulated image taken from Robot 1

The basic algorithm for determining pose is to first determine the relative angle the robot is in the image frame. This is accomplished by finding the vertical corner support beams of the robot. Assuming that three support beams can be seen, the differences in the distance between the two sets of lines (i.e. the left-center, and center-right sets of lines) will give an orientation. Using only this algorithm will result in a set of four ambiguous solutions, so another feature of the robot will have to be known to differentiate the ambiguity. For example, the vertical beams of robot with a square cross-section will look the same when oriented at intervals of 0, $\pi/2$, π, and $-\pi/2$, so another known feature will have to be exploited to de-conflict the possibilities. This analysis is required regardless because and least one unique feature must be known of all robots so they can differentiate between them. Once the orientation has been determined, the distance to the robot is computed from the relative size and the focal length of the camera. The image processing itself requires a pixel analysis of the entire image. In order to find three vertical support beams of the robot, background clutter must first be separated.

Figure 37 depicts the geometry involved in relating a robot of known size (square with length a) to the projection of that image on to the focal plane. In this situation, the camera with focal length f on Robot 1 is pointed straight ahead (up) and Robot 3 is in the field of view on the relative left of Robot 1. The values x_L, x_M, and x_R are found by the image processing that locates the three vertical support beams. From these 3 values and f, the relative bearing angles to each support beam ($\beta_L, \beta_M, \beta_R$) can

determined. Taking the relative pointing angle of the camera into account, the formula for the relative bearing of Robot 3 from Robot 1 is

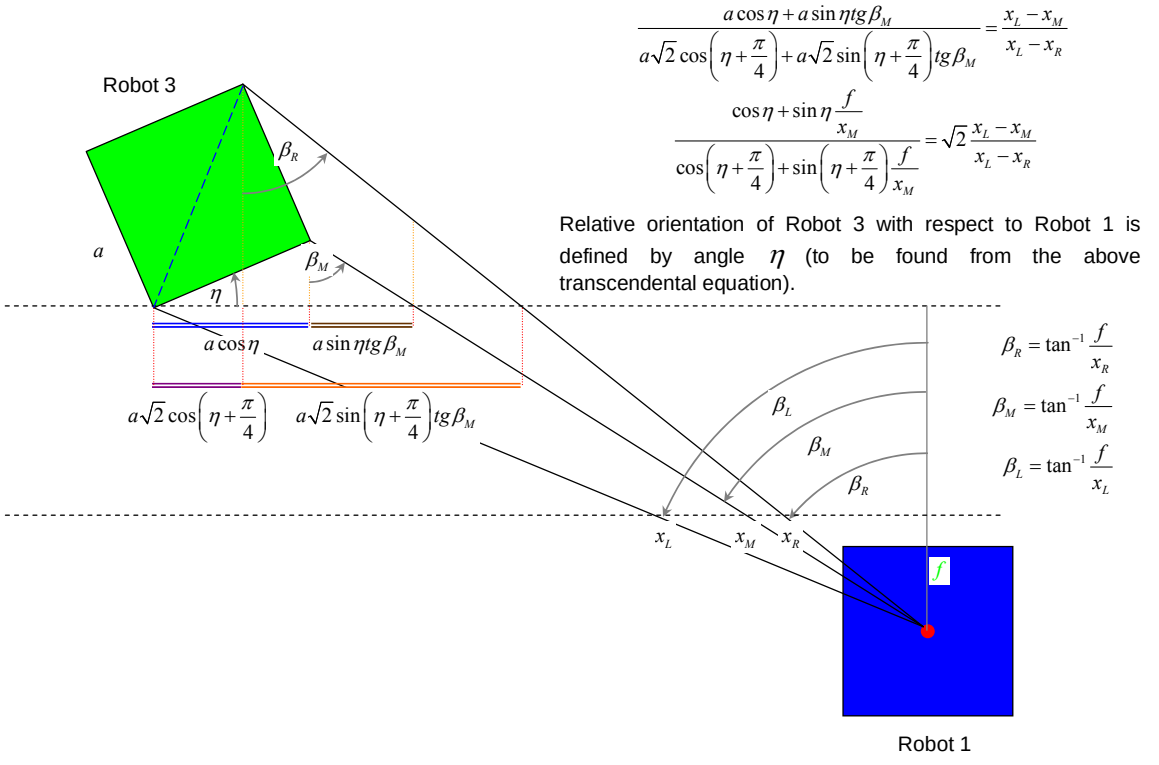
$$\beta = \alpha_{camera} + \frac{\arctan(f / x_R) + \arctan(f / x_L)}{2}$$

The orientation of Robot 3 on this bearing is described by η, which is found by solving the transcendental equation

$$\frac{\cos\eta + \sin\eta \dfrac{f}{x_M}}{\cos\left(\eta + \dfrac{\pi}{4}\right) + \sin\left(\eta + \dfrac{\pi}{4}\right)\dfrac{f}{x_M}} = \sqrt{2}\,\frac{x_L - x_M}{x_L - x_R}$$

Finally the range from Robot 1 to Robot 3 is determined. Since the camera will be mounted in the center of the robot, the range determined from the geometry in Figure 37 can be used. The equation used to determine the range is

$$R = a\,\frac{\sin\left(\dfrac{3\pi}{4} + \eta + \tan^{-1}\dfrac{f}{x_L}\right)}{\sqrt{2}\sin\left(\dfrac{\beta_L + \beta_R}{2}\right)}$$



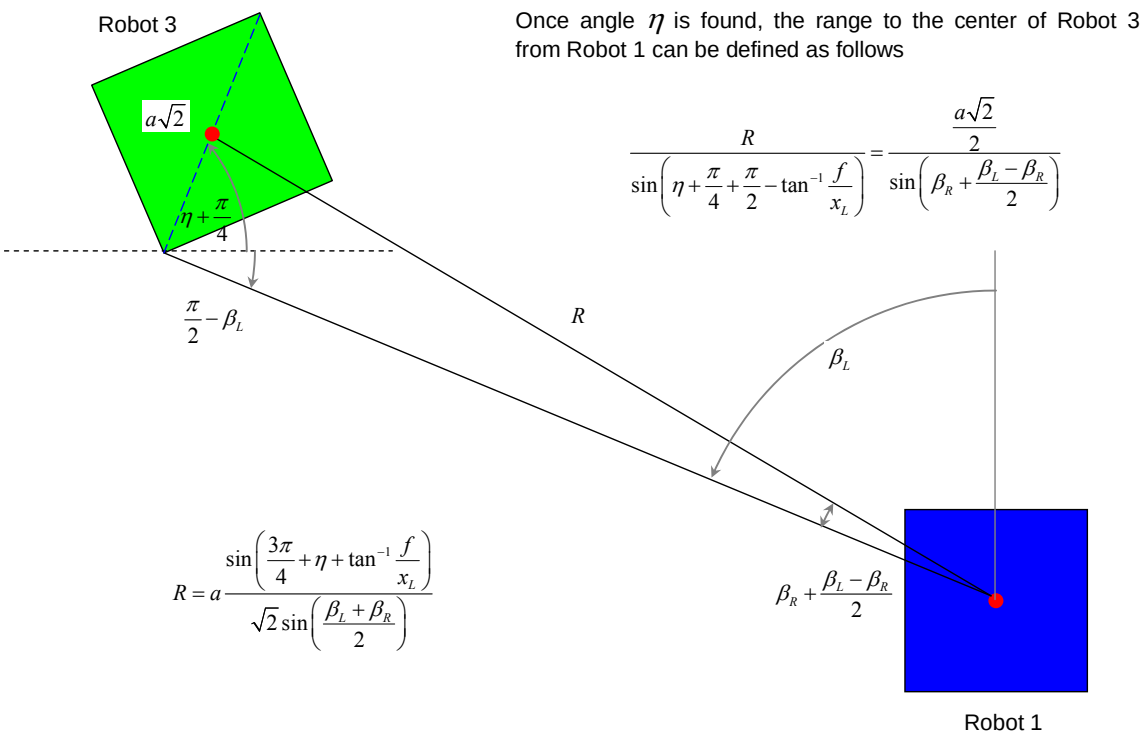
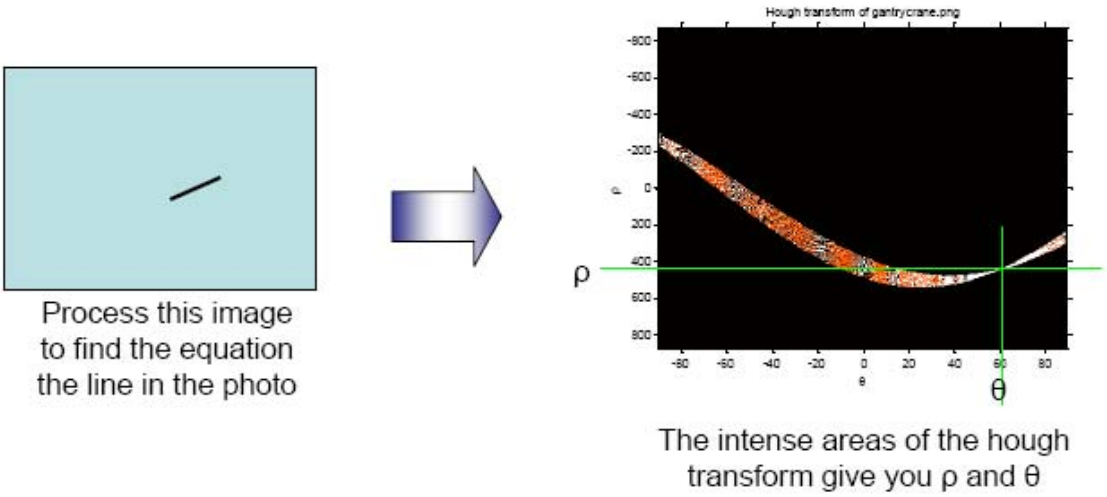


Figure 37. Pose estimation geometry for the leg supports

The Hough transform is a method for determining the equation for lines in a flat image. The MATLAB vision processing toolbox automates this process significantly with the hough(image) function. An example of how to use this function is depicted in Figure 38.



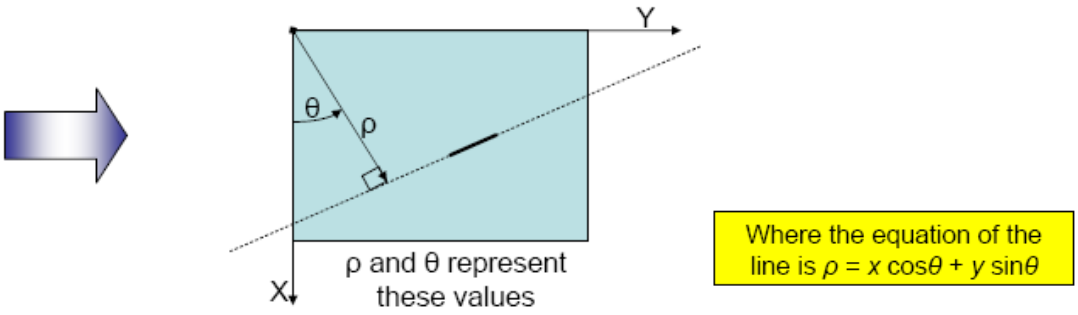


Figure 38. Hough transform example

The hough() function returns a matrix of values; the high values represent an index to a θ and ρ which can be used to define the equation of a line parametrically.

C. ONBOARD AUTONOMY

The navigation software also provides another function: onboard autonomy. The work completed here is meant to serve as a starting point, or platform, to develop robust guidance and control algorithms. Cooperative spacecraft conducting proximity operations will most likely need to operate autonomously in several different modes, such as when they are kilometers apart, meters apart, or centimeters apart. They would also benefit from being able to send each other messages, such as equipment status, or intentions. The navigation system will act as the brain controlling these functions; it is the ideal place to do it, as it will also be estimating the system state.

The navigation system is based around a finite state machine. It will consider the current state (from state estimation), the desired end state (from the user definition), any messages from the other robots (via the wireless LAN) and the finite state machine state variable to determine what the guidance mode should be, how the vision sensor may need to be controlled, and may also communicate any knowledge with the other two robots (via the wireless LAN). The navigation module is in Figure 39.

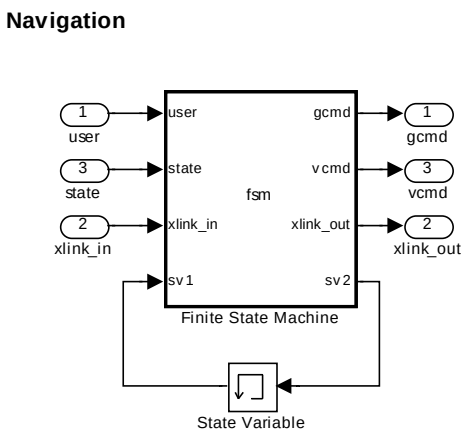


Figure 39. Onboard Autonomy SIMULINK model

An example finite state machine was developed as an initial step for robust navigation. This finite state machine commands the guidance system to not move (stay in position) until all of the robots have communicated that they are "ready" to invoke collaborative maneuvering (Figure 40). It is important to remember that this software is to run on all three robots; but in this case, states are named by their absolute names (Robot 1, Robot 2, and Robot 3), and not their relative names. This convention will limit which of the finite states each robot can go into.

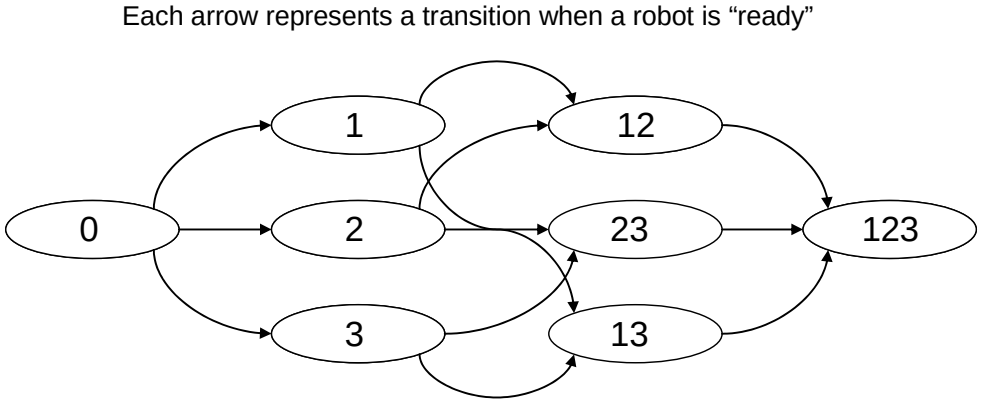


Figure 40. Example finite state machine of the onboard autonomy system

This diagram may be easier to understand in words. The transitions are explained assuming the perspective of Robot 1. Starting in State 0, all initialization routines are started. As each robot completes its initialization routine, it will send a message to the other machines and transition to the next state. In the case of Robot 1, the next state would be State 1, meaning Robot 1 is ready. Likewise, when the other robots received

the message that Robot 1 was ready, they would transition to State 1 as well. If Robot 2 was the next to finish initializing, the robots would all similarly transition to State 12, meaning Robots 1 and 2 are both ready. Finally, when Robot 3 sends its ready message to the other robots, the State 123 would be transitioned to and that represent all robots are ready to maneuver. Up to this point, the guidance modes on all robots would be commanded not to move the robots.

Other states can be added to this base finite state machine and robustness can be added with the addition of other messages, such as error messages, or the lack of messages, such as a lost communications scenario.

D. LIDAR

Paramount to the success of the AMPHIS experiment is the accurate determination of the system state. In order to enable autonomous operation of a multiple craft system, each craft needs a sensor to reliably access the positions, and to a lesser extent, the pose, of the other craft in the system. A LIDAR sensor was selected to be implemented first on AMPHIS because of it directly provides the two most critical parameters for the system state: bearing and range. It provides a good sample rate, and also requires less processing time than a photograph image. This chapter focuses on the actual hardware implementation of the SICK LD-OEM LIDAR. In contrast to modeling and simulation, this crucial part of the experimental setup has different types of problems that require the physical implementation, configuration, and operation of actual hardware.

1. SICK LD-OEM LIDAR

The SICK LIDAR uses a class 1 (eye safe) laser. Its primary capabilities and attributes are located in Table 4.

Connection types	RS232, CAN, ARCnet
Ranges	24 m (5% reflection) 50 m (20% reflection) 100 m (90% reflection) 250 m (with reflectors)

Size	10”x6”x5”
Weight	3.2 kg
Useful scanning angle	360˚
Operating Voltage	24V ± 20%
Power consumption	36W
Scanning Frequency	5 to 20Hz
Angular resolution	0.125˚
Max pulse frequency	14.4 kHz
Serial Data port baud rates	4800 to 115200 Bd

Table 4. SICK LIDAR OEM Product Information

2. LIDAR Setup

The SICK LIDAR requires two physical connections to operate: a power cable, and a control link cable. The power source must provide 19.2 to 28.8 V at 36 W. Having the correct power supply is crucial for reliable information. It is recommended from the vendor that a supply is used that is rated at twice the required 1.5A. For testing, a HP 6542A DC power supply rated at 0-20 V, 0-10 A was used to prevent the unnecessary recharging of the onboard battery, but the LIDAR is easily reconfigured between the two. For the control link, a RS-232 serial cable was connected to the COM port of a Windows XP Pentium III computer. The SICK LIDAR has a sample application to test and demonstrate the capabilities of the unit. Some example test runs are included here with illustrative screen shots to better depict the LIDAR capability.

3. LIDAR Control

The first step to controlling the SICK LIDAR was to communicate with it. MATLAB was selected for configuration setup and testing because of its simple, flexible, interface and integrated processing functions.

There are four basic functions that must be performed to operate the LIDAR via a COM port: a serial port connection must be opened and closed with the scanner, and data must be read from, and written to the scanner. The built in commands that directly

correlate to these functions in MATLAB are fopen(), fclose(), fread(), and fwrite(). Two other MATLAB functions, serial() and delete(), are used to identify the port to be opened, and delete the port when finished, respectively.

Although the use of the tools to communicate with the SICK LD-OEM LIDAR are straight forward, the composition and encapsulation of command data and subsequent decomposition, parsing, and interpretation of status/profile data are not. Although sending and receiving data to and from the scanner are similar, each is addressed separately to avoid confusion. But first, it is important to state that the data passed with fread() and fwrite() are always in bytes, or unsigned 8-bit integers (0-255, or 0x00-0xFF) represented in MATLAB by double precision floating point numbers. Depending on their position in the data stream, these integers may be converted to ASCII characters, hexadecimal values, or two bytes are combined to form 16-bit decimal values.

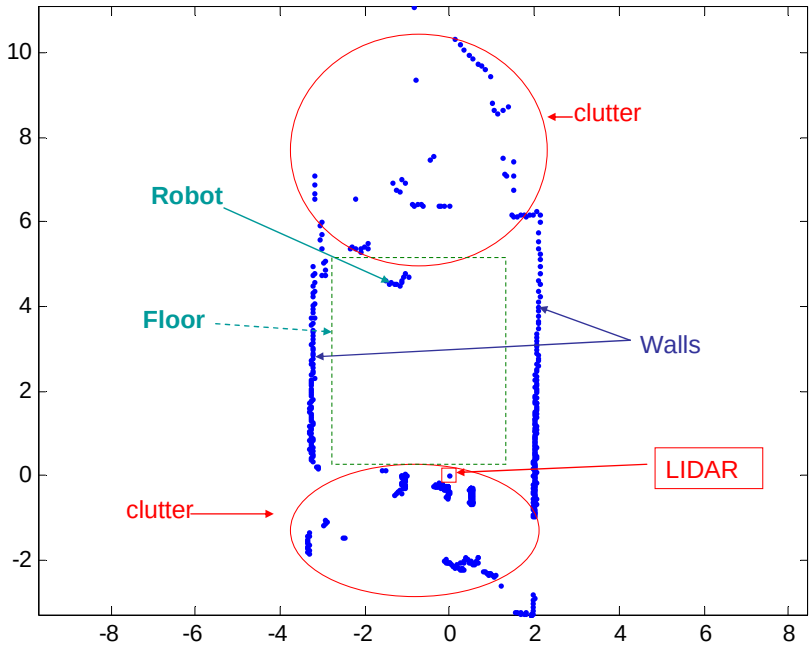


Figure 41. Illustrated output of the LIDAR

All data passed to the SICK LD-OEM LIDAR must be properly formatted into a "packet." A beginning of a packet is identified by the number 2, and the end of a packet is identified by the number 3. The bulk of the packet is composed of two parts: the command data (CD) and the Cyclic Redundancy Code (CRC). The order of significance of each packet sent is left to right: the leftmost number is sent first, the rightmost number

is sent last. Therefore, each packet sent to the scanner has the vector form [2 CD CRC 3], where CD is a 1xN vector (N can have several values based on the number of parameters of the command being sent), and CRC is a 1x4 vector. Although the length of CD can vary with that command being sent, all commands for the SICK LD-OEM LIDAR can fit into a single packet. The format of CD and CRC are discussed next, in order.

Command Data (CD) is the code sent to the scanner to control it. CD originates as a string of characters representing a hexadecimal code. It is important to stress that it is a string because each character in the string is converted to the ASCII value for that character before it is sent to the scanner. For example, the sub-string '0A' is not represented by the array [0 10], but by the array [48 65] (ascii('0') =48, ascii('A')=65). The command to convert a character string to an ASCII array in MATLAB is double('String') (which returns [83 116 114 105 110 103] for example). One major advantage of using this schema is that the command data streams sent and received by the scanner are limited to 16 integers, 48-57 and 65-60 ('0'-'9' and 'A'-'F'). These values make it easy keep packet header information (2 and 3) distinct from command data.

An example to illustrate the format for the packet/data structure is depicted in Figure 42. Note that the numbers shown in quotes are the strings that would be converted into an array of ASCII values as described above.

STX	SID	DID	LEN	SEP	CMD	PARM	CRC	ETX
2	'00'	'10'	'0004'	'0000'	'0403'	'0000'	'A3E3'	3

Figure 42. SICK OEM LIDAR Protocol stack

Portion	**Meaning**	**Note**
STX	Start of a packet	Always = 2
SID	Source identifier	'00' is the computer ID
DID	Destination identifier	'10' is the scanner ID

LEN	Number words remaining in the structure	In this case 4 represents one word each for SEP, CMD, PARM, and CRC
SEP	Separator – start of the command	Always '0000'
CMD	Command code	A list of primary commands follow
PARM	Parameters for the command	Number of parameters vary with command
CRC	Cyclic Redundancy Code	Calculated from all parts of the packet except STX, ETX, and CRC
ETX	End of a packet	Always = 3

Table 5. SICK OEM LIDAR Protocol Meaning

This example is the command to start the LIDAR spinning.

The Cyclic Redundancy Code (CRC) is a bit hash of the CD to ensure that it is genuine and was not received in error. The CRC is a 16-bit hexadecimal number that is calculated from the CD. It also must be converted to an ASCII array before it is sent to the scanner. Since the CRC is always 16-bits, the ASCII array that represents it is always four numbers in length. The CRC signature is calculated using the generator polynomial x16 + x12 + x5 + 1 as recommended by the ITU.T V.42 (former CCITT). A MATLAB implementation of the CRC calculator was written based on a C++ algorithm that came with the SICK LD-OEM LIDAR (Ref. [13]).

Every command sent to the scanner prompts a return of data from the scanner. For many commands, this returned data is simply a status of the command sent (success or failure). In these cases, the returned data can fit into a single packet, but the data length of a profile normally requires several packets to encompass the entire data stream. For this reason, synchronization between the scanner and the code controlling the scanner is essential. When scanner needs to send data through the serial port, it temporarily stores that data in a buffer on the buffer. When the fread() command gets the data from the buffer, the buffer is cleared to make room for more data. Profiles that are too large to fit in the buffer must be emptied promptly before new data overwrites the buffer.

To keep the scanner synchronized, MATLAB scripts or functions were written for every command utilized for the SICK LD-OEM LIDAR. Each command sent is immediately followed by a read command to check the status of the command sent and to keep the buffer clear. These read commands are completed sequentially, in serial, so program execution is held until either the read command is successful, or a scanner time-out error indicated that he scanner is not responding. Although it desired to be able to read data from the scanner in parallel with program execution, synchronization problems become too difficult to overcome. This problem exists because the scanner's internal buffer size is only 512 bytes. As data is written to the buffer, it must be read and cleared before the buffer is full. If the buffer is not cleared in time, the scanner will overwrite the buffer, and corrupt the results.

A read command is successful if it returns a packet (a data stream that starts with the number 2 and ends with the number 3). The packet must then be parsed to determine its meaning. The format for the packet/data structure is similar to that of the one depicted in Figure 42, with the following notes:

- The SID and DID will be swapped to indicate the dataflow in receiving data is opposite than sending data.
- The SEP field will normally contain '0000' to indicate a successful command competition, and 'FFFF' to indicate a command failure.
- The returned command code will be the same as the sent command code except the leftmost bit will be a 1 vice a 0 (in hexadecimal, the leftmost byte will be '8' instead of '0').
- All parameters values are returned.

If the packets being received are the result of a GET_PROFILE command, profile information is returned in multiple packets, with each packet containing a segment of the profile in the command fields of the packet structure (SEP, CMD, and PARM). These multiple segments must be gathered sequentially and then assembled to be processed. Figure 43 illustrates a profile that came in five segments (and therefore five packets). The first packet contains the 'FFFF' identifier to indicate the beginning of a profile. The next word indicates the number of segments that the profile will be divided into, and the rest of the data in the command fields contain the information asked for in the user specified profile definition. As each of the following segments are extracted

from the packet command fields, the first value indicates the segment number (which count down to 1) followed by more profile data. Since the profile indicator and segment numbers do not contain profile data, they are removed once they are checked for consistency. The profile data in all the segments are concatenated to be further examined.

STX	SID	DID	LEN	SEP	SEG	PRO	----	CRC	ETX
2	'10'	'00'	'0126'	'FFFF'	'0005'	####	####	####	3

STX	SID	DID	LEN	SEG	PRO	----	CRC	ETX
2	'10'	'00'	'0126'	'0004'	####	####	####	3

STX	SID	DID	LEN	SEG	PRO	----	CRC	ETX
2	'10'	'00'	'0126'	'0003'	####	####	####	3

STX	SID	DID	LEN	SEG	PRO	----	CRC	ETX
2	'10'	'00'	'0126'	'0002'	####	####	####	3

STX	SID	DID	LEN	SEG	PRO	----	CRC	ETX
2	'10'	'00'	'0056'	'0001'	####	####	####	3

Figure 43. SICK OEM LIDAR Protocol for Profile data

Finally, the profile data must be processed to be used. The first three words in a profile are standard: LD response, PROFILEFORMAT, and PROFILEINFO. PROFILEFORMAT should be the same value as set in the GET_PROFILE command. It is a bit pattern that indicates how to interpret the following information. The code '01B0' will be used normally because it returns the most data in the least amount space. The first six words give the number of points per sector, the starting direction for the sector, and the angle step for the sector, for each of two sectors. For scanner design reasons, at least two sectors must be specified even if only one sector contains points. In the case where PROFILEFORMAT = '01B0', the first sector has zero points and is ignored. The second sector therefore starts at 0˚, has an angle step of 0.625˚, and 575 points to give full 360˚ coverage. The following is an example of a GET_PROFILE command.

SEND:	00	10	0005	0000	0301	0001	01B0	13DD
Meaning:	SID	DID	LEN	SEP	CMD	NUM	INFO	CRC

The CMD is the code that represents a GET_PROFILE command. The parameters shown specify to get only one profile (NUM) in the format specified by ‘01B0’ (INFO).

LD response 8301

The response from the scanner, ‘8301’, is in response to ‘0301’ as described above.

PROFILEFORMAT 01b0

The PROFILEFORMAT is in the desired format.

PROFILEINFO 01 02

The first byte is aways ‘01’. The second byte means there are two sectors. The

SEC1: Angle step 0.625 deg

SEC1: Number of points of sector 0

SEC1: Start direction of 359.375 deg

SEC2: Angle step 0.625 deg

SEC2: Number of points of sector 575

SEC2: Start direction of 0.000 deg

All code developed to control the LIDAR is included in the Appendix.

V. ON ORBIT APPLICATIONS

A. ON-ORBIT COMPARISONS (HILL'S EQUATIONS / CLOHESSY-WILTSHIRE EQUATIONS)

Only three degrees of freedom (DOF) are considered instead of 6 DOF that a rigid-body spacecraft would have. In fact, the robots are considered to move along a leveled surface. This simplification limits comparisons of these ground based experiments to operations on orbit to cases where motion between craft is in the same orbit plane, and each craft can maintain its orientation constant relative to the orbit plane. This limitation is acceptable and in line with most current concept of operations for orbital rendezvous; the Space Shuttle, for example, completes all rendezvous maneuvers with the International Space Station in a single orbit plane. (Ref [12]) The other major simplifications are the weightless environment of orbit flight, which is impossible to recreate in three dimensions in a laboratory environment. However, the special friction-free floor approximates weightlessness in translational movement, and computer simulation of Hill's equations, or other model, can be implemented to approximate differences between ground and on-orbit operations. (Ref [10]). In other words, application of the AMPHIS 3 DOF simulator to test and evaluate 6 DOF systems presupposes that the spacecraft can sense, control, and maintain its pitch, roll, and out-of-plane distance. Assuming that these three degrees of freedom and controlled to be constant, the remaining three degrees of freedom can be simulated on the AMPHIS test bed. Of these remaining three degrees of freedom, the rotation about the vertical axis is decoupled from the other two (translation in the orbit plane), and the dynamics of the translation in the orbit plane can be expressed easily using Hill's equations (also known as the Clohessy-Wiltshire (CW) equations). The following discussion, however, will not be limited to in-plane motion.

Hill's equations describe the relative movement of a "deputy" satellite to a "chief" satellite, or hypothetical mass in orbit, in three dimensions. A set of equations can be derived if some simple assumptions are made (Ref. [11]):

- The chief is in a circular orbit with radius a.
- The deputy is relatively close to the chief (ρ<<a).
- There are no perturbations.

With these assumptions, the position of the deputy can be approximated by the following equations:

$$x = -(2v_0 / n + 3x_0)\cos\psi + (u_0 / n)\sin\psi + 4x_0 + 2v_0 / n$$

$$y = (y_0 - 2u_0 / n) + (4v_0 / n + 6x_0)\sin\psi + 2u_0 / n\cos\psi - (6x_0 + 3v_0 / n)\psi$$

$$z = z_0\cos\psi + (w_0 / n)\sin\psi$$

These equations express the position of the deputy in the RSW coordinate system. This three-dimensional, orthogonal coordinate system is defined with the origin at the position of the chief, the R axis points directly away from the center of the earth, the S axis points in the direction of the instantaneous velocity of the chief, and the W axis completes the right hand rule. Therefore, the in-plane translation is represented by x and y, and the out-of-plane position is represented by z (Ref [10]).

The time rate of change of the true anomaly, $n = \sqrt{\frac{\mu_\oplus}{a^3}}$ and $\psi = n(time)$.

The velocity terms ($[\dot{x}, \dot{y}, \dot{z}]^T = [u, v, w]^T$) are found by taking the derivatives of the position terms:

$$u = (2v_0 + 3nx_0)\sin\psi + u_0\cos\psi$$

$$v = (4v_0 + 6nx_0)\cos\psi - 2u_0\sin\psi - 6nx_0 - 3v_0$$

$$w = -nz_0\sin\psi + w_0\cos\psi$$

Notice that the z component (out-of-the-orbit plane) is completely decoupled from the others (in-the-orbit plane), and all the equations are functions only of time, the orbital radius, and the initial conditions. For this reason, a real-time simulator was implemented rather easily in MATLAB. Figure 44 illustrates the reference frame.

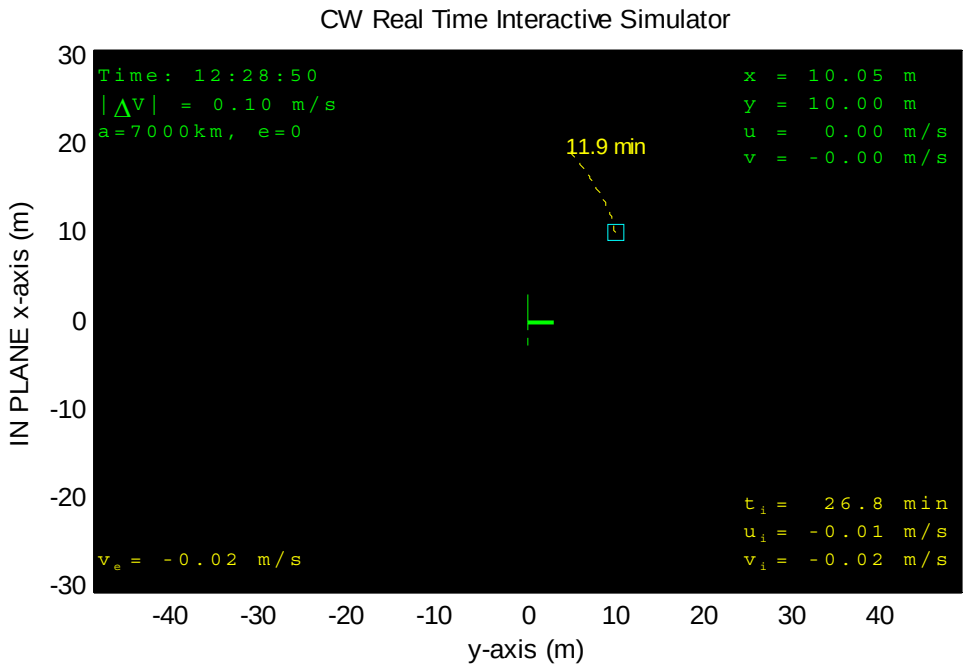


Figure 44. CW Reference frame

The following bullets describe the figures used in this section.

- Center (green axis) = The chief (reference spot)
- The heavy line is the velocity vector
- The green dotted line points to the earth
- The blue star is the center of the earth (seen in the y-z plane, Figure 51)
- Up direction = positive x axis (in-plane, altitude) opposite the radius vector
- Right = positive y axis (in-plane, in-track) parallel to the chief velocity vector
- Into the paper = positive z axis (out of plane, off track)
- Cyan square = position of the deputy
- Yellow dotted line = predicted motion based on current velocity
- The distance from the deputy to the chief is defined as ρ

There are five different views available in the simulator:

- XY plane (in-plane motion)
- YZ plane (out-of-plane motion)
- 3-D stationary
- 3-D rotating (with respect to the earth)
- 3-D rotating and translated

A real-time simulator can be beneficial over a complicated model. A SIMULINK model that calculates multiple perturbations and integrates over a variable time slice can take much longer than real time, and you cannot interact with it. A real-time simulator can calculate positions based solely by the system clock and Hill's equations. If you can live without the accuracy, a real-time simulator can be very illustrative of on orbit proximity operations.

1. Real-time Simulator Basics

The CW Real Time Interactive Simulator (CWRTIS) developed for thesis research was motivated by NASA's Rendezvous Proximity Operations Program (RPOP). The crew of the Space Shuttle uses this software on a laptop (Payload General Support Computer (PGSC)) to dock with the International Space Station. Figure 45 is a screen shot from the RPOP program. On-orbit proximity operations can be counter-intuitive considering relative motion of two objects in slightly different orbits. Since RPOP was employed several years ago, crew performance in accurately and safely docking the Shuttle to the ISS has significantly increased (Ref. [12]).

CWRTIS assumes that the deputy does not use continuous thrust, as the Space Shuttle does not. Instead, it approximates thrusts to be infinitely short and produce a perfectly described change in velocity (ΔV). This is a good approximation considering the fidelity of the model. The amount of thrust to be applied can be changed and ranges from 0.01 m/s and up. The numeric keypad controls the direction of the thrust in terms of the x-y axis, or the reference frame described in Figure 44. The keys 8, 4, 6, and 2 are intuitively placed and represent a ΔV in the up, left, right, and down direction respectively. The diagonal keys (7, 9, 1, and 2) include a ΔV in two directions (also

intuitively placed). The 0 key is set to stop all relative motion (or, create an equal ΔV in the direction opposite of the current velocity). The 5 key will boost the current velocity by a factor of ΔV. In thrusting with the keypad, the deputy's motion and velocity are changed.

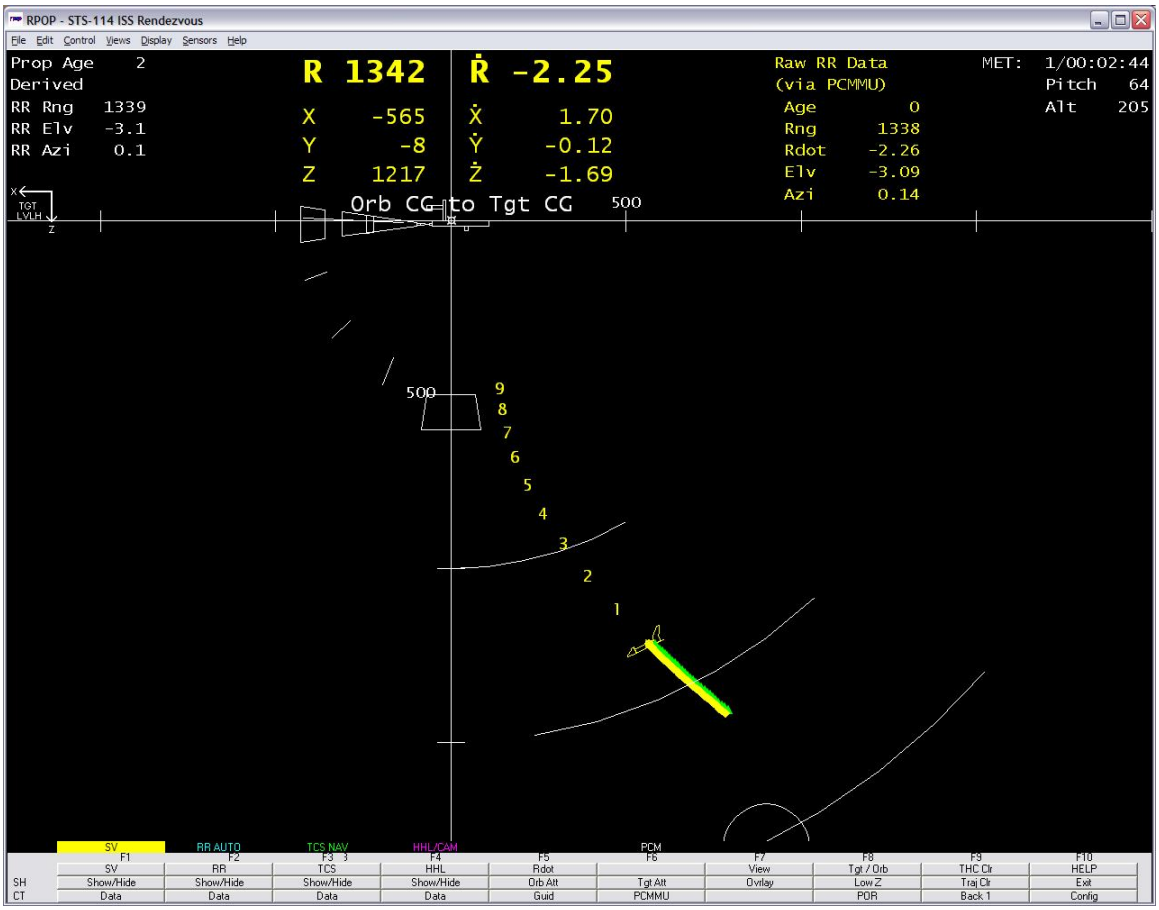


Figure 45. Screen shot from NASA's RPOP (Ref. [12])

The yellow-dotted line in Figure 44 predicts the motion of the deputy if no additional thrusts are made. The length of this predictor can be interactively changed. The default length is one orbit period. Figure 46 shows the predicted motion of the deputy for approximately two orbits.

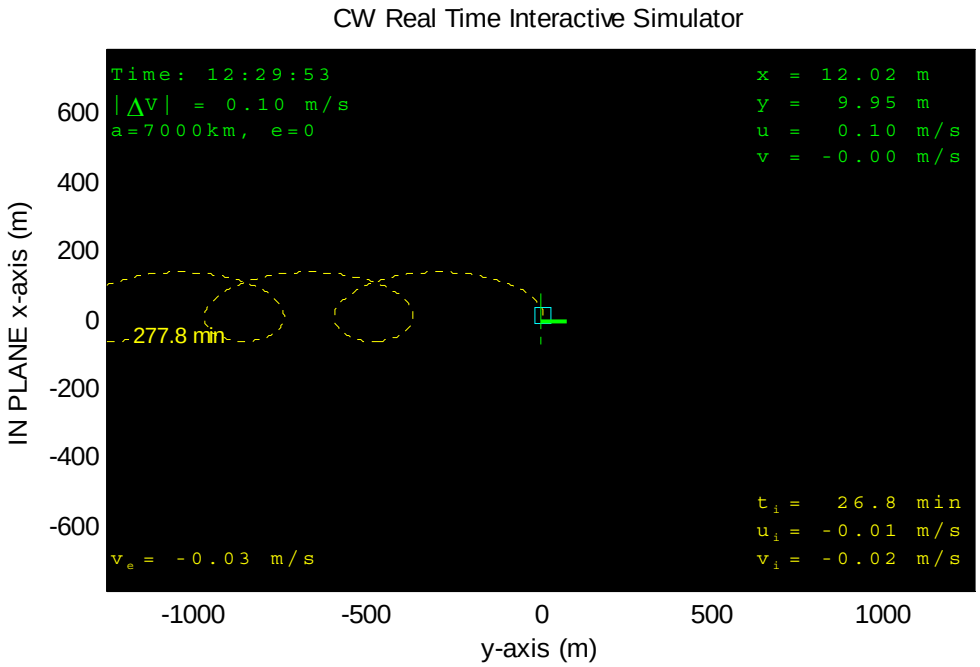


Figure 46. Predicted motion for 2-3 orbits

2. Interception Problem

Vallado derives the intercept equations based on the Hill's equations (Ref [10]). Given a "time to intercept" the following equations provide the initial velocity required to intercept the chief at the prescribed time.

$$v_0 = \frac{(6x_0(\psi - \sin\psi)) - y_0)n\sin\psi - 2nx_0(4-3\cos\psi)(1-\cos\psi)}{(4\sin\psi - 3\psi)\sin\psi + 4(1-\cos\psi)^2}$$

$$u_0 = -\frac{nx_0(4-3\cos\psi) + 2v_0(1-\cos\psi)}{\sin\psi}$$

$$w_0 = -z_0 n\cot\psi$$

Once derived, these equations are rather straight forward. Notice again the decoupling of the out-of plane motion. Implementing these equations in CWRTIS validates the equations. Arbitrarily, if the time to intercept (in seconds) is set as the distance from the chief to the deputy (ρ in meters), the closure rate that results is about 1 m/s. Figure 47 depicts the predicted motion of the deputy to rendezvous with the chief after the velocity was set using the intercept equations (employed by pressing the period key).

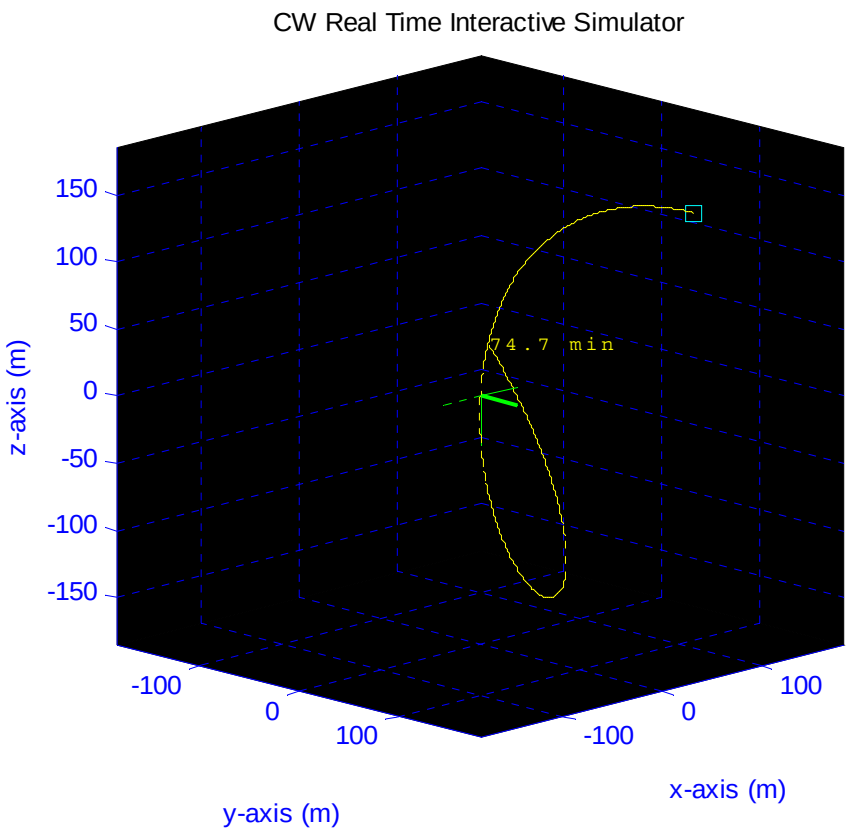


Figure 47. Rendezvous trajectory

Notice that the trajectory does not necessarily have to be in-plane motion. Interesting to note, the magnitude of the velocity to intercept has a non-intuitive relation to the time to intercept: increasing the time to intercept does not necessarily decrease the magnitude of the velocity required. This simulator actually chooses a time to intercept using the minimum velocity required to intercept as determined by the MATLAB function `fminsearch`. This time and velocity appears in yellow at the bottom right hand corner of views 1 and 2.

3. Relative Motion Obits

Alfriend demonstrates that by selectively choosing the in track velocity, the relative motion orbit can be a stationary 2x1 ellipse (Ref. [11]). If $v = -2nx$, the relationship between x and y become constant: $x^2 + y^2/4 = const$, which is an ellipse. The size and location of the ellipse is based on the other variables (z,u,w,n). Figure 48 depicts the relative orbit of the deputy about the chief.

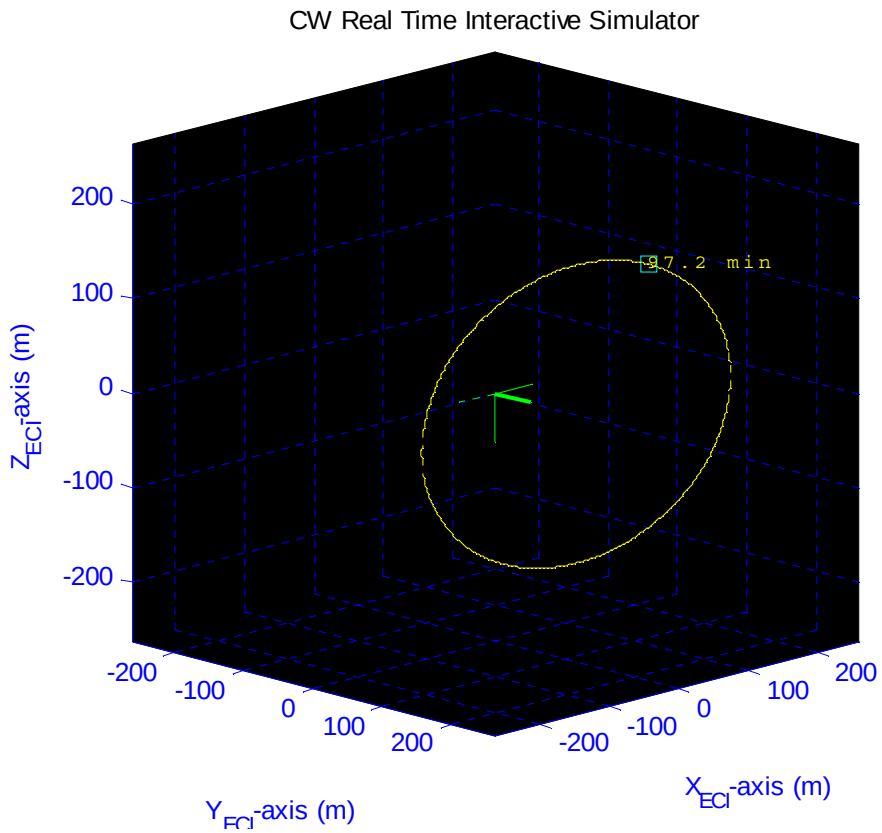


Figure 48. Elliptical Relative Orbit

Interestingly, the size and orientation of the ellipse can be changed by thrusting in the x and z directions. Thrusting in the *v* direction will disrupt the aforementioned relationship and therefore the stationary ellipse. Alfriend further derives Hill's equations in terms of the orbital elements so these relative motion orbits can be more easily implemented to real orbits (Ref. [11]). Theoretically these orbits can be maintained with little fuel; however some elliptical relational orbits are too sensitive to perturbations to be practical. Again, this simulation ignores perturbations, so that that analysis is not accomplished here.

Also described by Alfriend are circular orbits relative to orbit plane and projected on the YZ plane. These orbits are examined by looking at the general solutions to Hill's equations listed in Vallado (in slightly different notation) (Ref. [10], [11]):

$$x - x_c = C\sin(nt + \psi_0),\ y - y_{c_0} = \dot{y}_c t + 2C\cos(nt + \psi_0),\ z = D\sin(nt + \varphi_0)$$

The constants C and D are expressed as:

$$C = \sqrt{\left(3x_0 + \frac{2v_0}{n}\right)^2 + \left(\frac{u_0}{n}\right)^2}\ ,\ D = \sqrt{\left(\frac{w_0}{n}\right)^2 + z_0^2}$$

The center point about which the relative orbit centers upon is denoted as (x_c, y_c, z_c). It is possible that the y term actually moves. The relationships of this center point are derived in Vallado (Ref. [10]):

$$x_c = 4x_0 + \frac{2v_0}{n} = \frac{-2\dot{y}_c}{3n}$$

$$y_{c_0} = y_0 - \frac{2u_0}{n}$$

$$\dot{y}_c = -6nx_0 - 3v_0$$

$$z_c = z_0\ ,\ \varphi_0 = \psi_0$$

Choosing a center point of (*0,0,0*) and $\dot{y}_c = 0$, the following initial conditions ($t_0 = 0$, $\psi_0 = 0$) are found:

$$x_0 = 0$$

$$y_0 = 2C$$

$$z_0 = 0$$

$$u_0 = \frac{ny_0}{2} = nC$$

$$v_0 = -2nx_0 = 0$$

$$w_0 = n\sqrt{D^2 - z_0^2}$$

Choosing $D = \sqrt{3}C$ provides the following relationship:

$$x^2 + y^2 + z^2 = const = C^2 \sin^2 + 4C^2 \cos^2 + 3C^2 \sin^2 = 4C^2$$

This represents a circular orbit on a sphere centered at (*0,0,0*). The size of the circle is specified by choosing C accordingly (Figure 49).

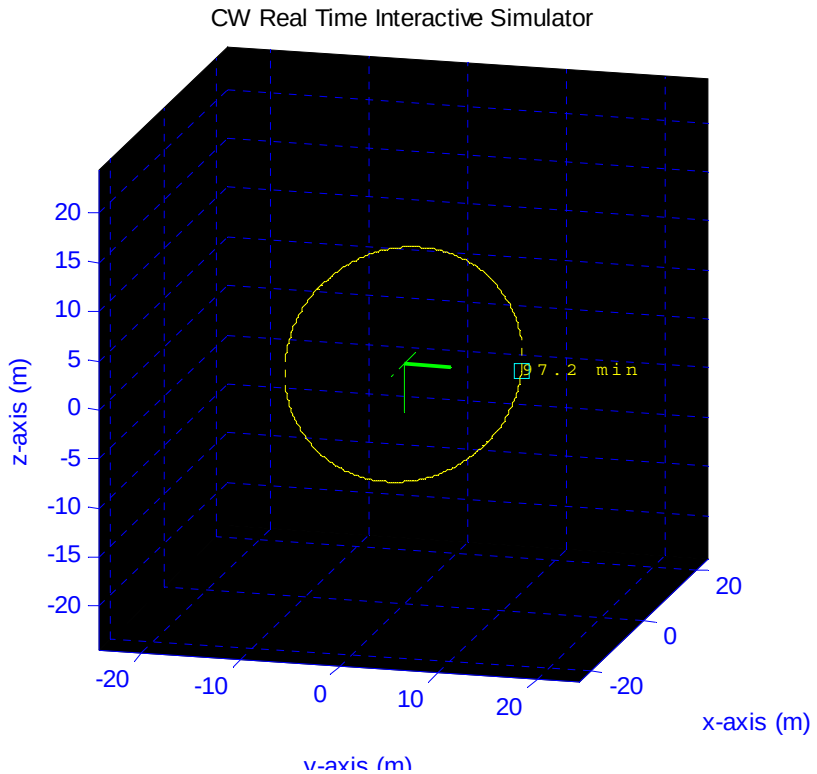


Figure 49. Circular Orbit on a Sphere centered at (*0,0,0*)

Choosing $D = 2C$ provides the following relationship:

$$y^2 + z^2 = const = 4C^2 \cos^2 + 4C^2 \sin^2 = 4C^2$$

This represents a circular orbit as projected on the YZ plane. Again, the size of the circle is specified by choosing C accordingly.

By changing the initial phase angle for the z term only ($\psi_0 \neq \varphi_0$), it is also possible to select initial conditions that achieve a circular orbit projected onto the XZ plane as well. The XY plane, however, are coupled and will always have a 2x1 relationship for elliptical relative orbits.

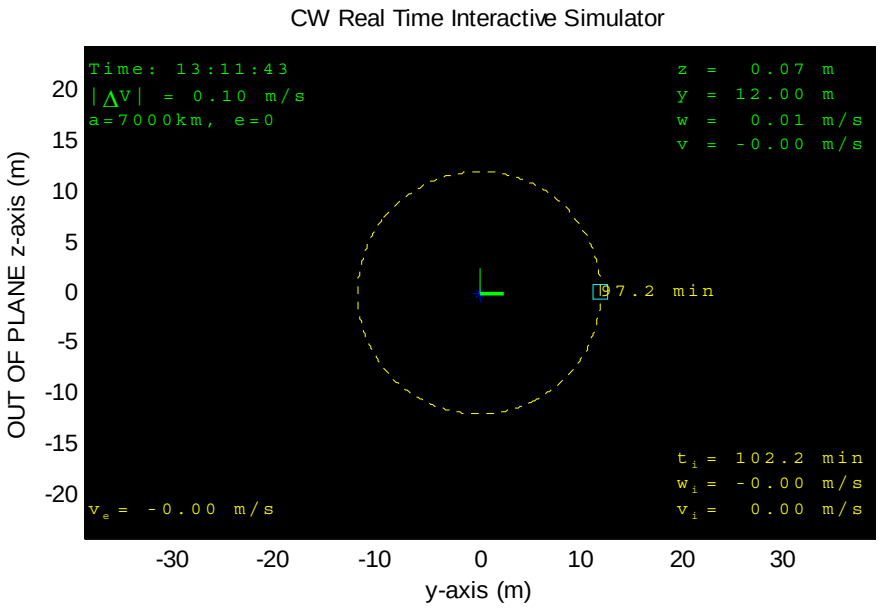


Figure 50. Circular Orbit Projected onto the YZ plane

Finally, consider a case were the center-point of the orbit is not at (*0,0,0*). Choosing a center-point of (*1,1,1*) creates a y-drift term: $\dot{y}_c < 0$. Looking at the trajectory of the relative orbit in the ECI frame illustrates that the motion is not elliptical. To magnify the effect, an impractical orbit of 70 km is chosen. This goes against the assumptions made for Hill's equations to be accurate, a>>ρ, but it is an illustration of what is happening mathematically. On a more realistic scale, these effects would appear much smaller; however, they will still be there. Since this orbit is not truly elliptical, it cannot be employed without the use of using corrective thrusts. Figure 52 illustrates the features implemented in CWRTIS.

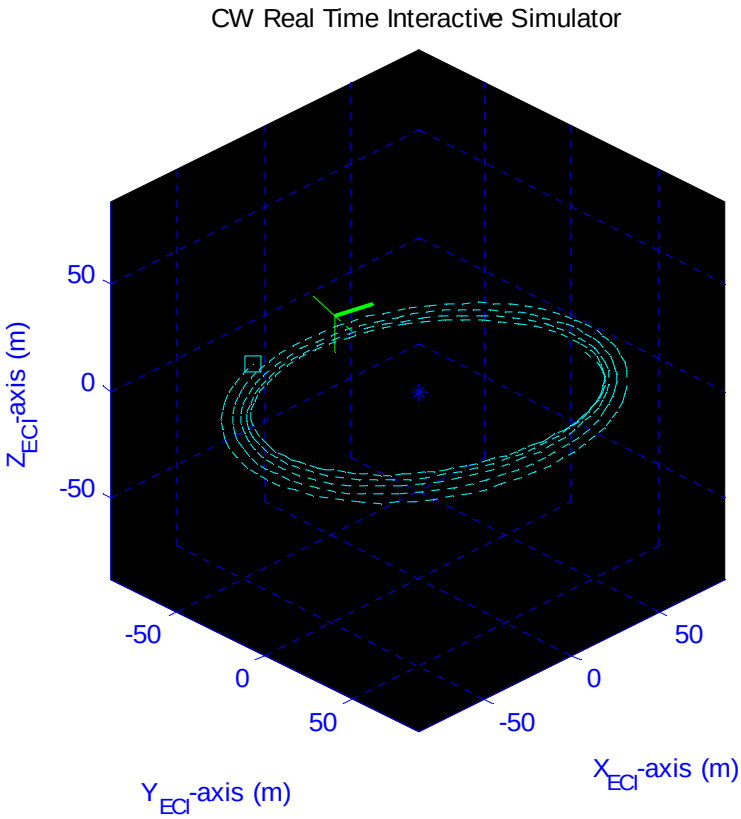


Figure 51. Relative Motion in the ECI frame (a=70km, $\dot{y}_c < 0$)

CW Real Time Interactive Simulator

number pad: 1-9	activate in-plane thrust
+/-	+/- out-of-plane thrust
0	stop relative motion
shift-[/]	+/- ΔV amount
shift-1/2/3/4/5	display mode
p	on/off predictor
[/]	+/- predictor length
c	on/off center mark
*	orbit in ellipse
.	intercept
</>	+/- zoom
,	auto-zoom
?	help screen
q	quit
ΔV used thus far	**0.22 m/s**

Help Screen

Figure 52. CWRTIS Help Screen

4. Applying Hill's Equations to the AMPHIS Test Bed

Applying Hill's equations to the AMPHIS test bed can be accomplished easily in simulation, but would not be practical in hardware. Using only the basic Hill's equations, the plant dynamics could be changed to simulate relative motion behavior. (Ref. [10])

Change from:	To:
$\ddot{x} = f_x$	$\ddot{x} = f_x + 2\omega\dot{y} + 3\omega^2 x$
$\ddot{y} = f_y$	$\ddot{y} = f_y - 2\omega\dot{x}$

Thrusts are denoted by *f*. The orbital angular rate, ω, is a constant which depends on orbital altitude and the mass of the earth. These equations express the relative accelerations from the (0,0) position of the floor, but could be modified to be centered about any part of the floor. All code for the CWRTS is included in the Appendix.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS

A. SUMMARY

This research covers several topics within the SRL's development of AMPHIS: Autonomous, Multi-Agent, Physically Interacting Spacecraft simulator test bed. The software architecture has been developed. A major portion of the software needed to run onboard the simulator robots was developed in the MATLAB/SIMULINK environment. This software can serve as a simulator, or it can be easily configured to run on multiple platforms. Furthermore, it is modularized to facilitate easy plug-and-play testing of newer algorithms, and can simulate any portion of a scenario instead of using hardware. This design enables hardware in-the-loop testing.

Several types of GNC algorithms were developed to test and validate the software. First, a trajectory planning algorithm that used the Direct Calculus of Variation Algorithm was developed using a single camera, single thruster configuration. Second, an Artificial Potential Function Guidance was developed to evaluate a dynamically updating algorithm. The bulk of the software was also validated during hardware tests.

The usefulness of LIDAR was explored in the course of this thesis. This research included the control of the LIDAR; the parsing and decoding of the data retrieved from the LIDAR; transforming LIDAR data into an absolute coordinate frame to help determine robots from other object in the room; and obtaining pose estimation information from the LIDAR data.

Onboard autonomy was examined with the development of an initial finite state machine. The concepts mentioned for the simple, studied case can be expanded to accommodate more robust systems. Several pose estimation strategies were also developed using digital imagery cameras. Finally, on-orbit application of a 3-DOF simulator in a 6-DOF environment was studied through the development of a real time, relative motion simulator.

B. FUTURE WORK

This research concludes as other research can begin. The following is a list of further research topics that could accelerate the success of the AMPHIS testbed.

- Make the hardware setup more robust and more reliable. More research in this area could facilitate better results throughout the rest of the system.
- Replicate the prototype robot two more times so interactive testing can begin.
- Improve the iGPS setup. The iGPS works well with its own software, however, this software does not integrate well with other things. To get real time iGPS updates, a better configuration that may involve directly decoding the iGPS receiver signal would improve the overall system setup.
- Add more states to the finite state machine in the onboard autonomy. Once there are several robots that can interact, it would be advantageous for multiple guidance algorithms to change autonomously, based on the current state. For example, the robot could go automatically from a positioning phase to a docking phase.
- Develop a better state estimator. The current state estimator is deterministic; it could be greatly improved with a Kalman filter. A way to easily enable/disable sensors for different scenarios would also be helpful.
- Port the software to LINUX/RT. Real time LINUX could most probably provide all the function needed for the robot on one CPU. Although this solution would be more elegant, is would displace the ease of modification that MATLAB provides.
- Develop better algorithms. Once the aforementioned improvements are made to the testbed, it can operate as it was intended to: to perfect GNC algorithms!

APPENDIX: MATLAB CODE

This table contains an index to the code presented in the Appendix. For future work on the AMPHIS project, these files can be found in the SRL shared directory under "\\Special.ern.nps.edu\srl$\BlakeEikenberry\Thesis Code."

1) Pose Estimation from Points Code
    - testbed.m
    - makexfrom.m
    - poseest.m
    - poseesttest.m
    - posexform.m
    - simphoto.m
    - plotobj.m
2) Direct Calculus of Variation Method Code
    - computetraj.m
    - draw_traj.m
3) AMPHIS xPC Target Artificial Potential Function Guidance Code
    - apfg.m
4) AMPHIS xPC Target Initialization Code
    - global_props.m
    - initialize.m
    - pe2st.m
    - portConfig.m
    - THRUSTSIMINIT.m

5) AMPHIS Animation Code

- anim_floor.m
- draw_dev.m
- draw_floor.m
- draw_foto.m
- draw_robot.m

6) AMPHIS Windows XP Related Code

- closeport.m
- hex2decword.m
- LidarBaud.m
- LidarCRC.m
- LidarGetprofile.m
- LidarIdlem
- LidarInit.m
- LidarMeasure.m
- LidarMeasureStop.m
- LidarParse.m
- LidarProfile.m
- LidarRead.m
- LidarSpin.m
- LidarTest.m
- openport.m
- plotProfile.m
- poseProfile.m

- readdata.m
- writedata.m

7) Clohessy-Wiltshire Real Time Simulator Code

- CW.m
- userevent.m
- findminV.m
- plotorb.m

```
%% testbed.m
%  this is the test bed for testing a pose estimation algorithm using
%  points
%  Written by LCDR Blake Eikenberry, 2005-2006

clc; close all;
fl=1; % focal length

%  define figure
obj.name = 'triangle';
obj.points=[-.5 -.5  .5 ;
            -.5  .5  .5 ;
              0   0   0];
obj.path=[1 2 3 1];

makexform(obj)

% test pose - try different poses
rot=[45,-20,40]; % degrees
pos=[1,-3,10];

[err, Pe]=poseesttest(obj,[rot pos],fl)

%  define figure
obj.name = 'square';
obj.points=[-.5 -.5  .5  .5;
            -.5  .5  .5 -.5;
              0   0   0   0];
obj.path=[1 2 3 4 1];



%% poseesttest.m
%  this function takes an object in 3-space, projects it onto a focal plane
%  adds noise, and then tries to estimate the pose of the object using only
%  the noisy projection in 2-space.
%  Written by LCDR Blake Eikenberry, 2005-2006

function [err, Pe]=poseesttest(obj,X,fl)
```

```
% get simulated image points
X(1:3)=[deg2rad(X(1:3))];
[p,P]=simphoto(obj, X, fl);

% p=simulated points on image
% P=actual points in space; the unknown
plotobj(obj,P,3,'b'); hold on; plotobj(obj,p,3,'b:');
figure; plotobj(obj,p,2,'b');

p(3,:)=[]; p=p(:);  % reshape points to match non linear functions

Xi=[0,0,0,0,0,20];  % initial guess
Xe=poseest(obj, p, Xi, fl);

[pe,Pe]=simphoto(obj,Xe,fl);

hold on; plotobj(obj,pe,2,'r:');

err=Xe-X;


%% makexfrom.m
%  this function wil creat the proper transform to project it
%  from 3-space to 2-space
%  Written by LCDR Blake Eikenberry, 2005-2006

function s=makexform(obj)
syms x y z a b g f real
pos=[x;y;z]; % position
rot=[a,b,g]; % rotation
Rabg = [cos(g) -sin(g) 0; sin(g) cos(g) 0; 0 0 1]*...
       [cos(b) 0 sin(b); 0 1 0; -sin(b) 0 cos(b)]*...
       [1 0 0; 0 cos(a) -sin(a); 0 sin(a) cos(a)];
P=Rabg*obj.points;
% translate the object
for i=1:length(P)
    P(:,i)=P(:,i)+pos;
end
clear s;
for i=1:length(P);
    s(:,i)=simple([P(1,i)*f/P(3,i);P(2,i)*f/P(3,i)]);
end
s=s(:);



%% posexform.m
%  this function perfroms the 3-D to 2-D transfrom on a paerticular object
%  Written by LCDR Blake Eikenberry, 2005-2006

function X=posexform(v,p,f)
a=v(1); b=v(2); g=v(3); x=v(4); y=v(5); z=v(6);
X=[

[ -(cos(g)*cos(b)-sin(g)*cos(a)+cos(g)*sin(b)*sin(a)-2*x)*f/(sin(b)-
cos(b)*sin(a)+2*z)]
[ -(sin(g)*cos(b)+cos(g)*cos(a)+sin(g)*sin(b)*sin(a)-2*y)*f/(sin(b)-
cos(b)*sin(a)+2*z)]
[ (-cos(g)*cos(b)-
sin(g)*cos(a)+cos(g)*sin(b)*sin(a)+2*x)*f/(sin(b)+cos(b)*sin(a)+2*z)]
[ (-
sin(g)*cos(b)+cos(g)*cos(a)+sin(g)*sin(b)*sin(a)+2*y)*f/(sin(b)+cos(b)*sin(a)+2
*z)]
```

```
[ -(cos(g)*cos(b)-sin(g)*cos(a)+cos(g)*sin(b)*sin(a)+2*x)*f/(sin(b)-
cos(b)*sin(a)-2*z)]
[ -(sin(g)*cos(b)+cos(g)*cos(a)+sin(g)*sin(b)*sin(a)+2*y)*f/(sin(b)-
cos(b)*sin(a)-2*z)]



];
X=X-p;

return

% square
[ (cos(g)*cos(b)-sin(g)*cos(a)+cos(g)*sin(b)*sin(a)-2*x)*f/(-
sin(b)+cos(b)*sin(a)-2*z)]
[ (sin(g)*cos(b)+cos(g)*cos(a)+sin(g)*sin(b)*sin(a)-2*y)*f/(-
sin(b)+cos(b)*sin(a)-2*z)]
[ (-cos(g)*cos(b)-
sin(g)*cos(a)+cos(g)*sin(b)*sin(a)+2*x)*f/(sin(b)+cos(b)*sin(a)+2*z)]
[ (-
sin(g)*cos(b)+cos(g)*cos(a)+sin(g)*sin(b)*sin(a)+2*y)*f/(sin(b)+cos(b)*sin(a)+2
*z)]
[ -(cos(g)*cos(b)-sin(g)*cos(a)+cos(g)*sin(b)*sin(a)+2*x)*f/(sin(b)-
cos(b)*sin(a)-2*z)]
[ -(sin(g)*cos(b)+cos(g)*cos(a)+sin(g)*sin(b)*sin(a)+2*y)*f/(sin(b)-
cos(b)*sin(a)-2*z)]
[ -(cos(g)*cos(b)+sin(g)*cos(a)-
cos(g)*sin(b)*sin(a)+2*x)*f/(sin(b)+cos(b)*sin(a)-2*z)]
[ (-sin(g)*cos(b)+cos(g)*cos(a)+sin(g)*sin(b)*sin(a)-
2*y)*f/(sin(b)+cos(b)*sin(a)-2*z)]
```

```
%% simphoto.m
%  this function takes an object in 3-space and plots it's projection
%  onto a 2-D plane
%  Written by LCDR Blake Eikenberry, 2005-2006

function [p,P]=simphoto(obj, X, f)
pos=X(4:6)'; % position
rot=X(1:3); % rotation

a = rot(1); b = rot(2); g = rot(3);
Rabg = [cos(g) -sin(g) 0; sin(g) cos(g) 0; 0 0 1]*...
       [cos(b) 0 sin(b); 0 1 0; -sin(b) 0 cos(b)]*...
       [1 0 0; 0 cos(a) -sin(a); 0 sin(a) cos(a)];
P=round(Rabg*obj.points*1e2)/1e2;
% translate the object
for i=1:length(P)
    P(:,i)=P(:,i)+pos;
end
p=[];
for i=1:length(P);
    p(:,i)=[P(1,i)*f/P(3,i);P(2,i)*f/P(3,i);f];
end



%% poseest.m
%  this function take an object defined by corner points
```

```
%  and it's projection on a 2-D plane and tries to estimate it's
%  pose in 3-space
%  Written by LCDR Blake Eikenberry, 2005-2006

function [Xe]=poseest(obj, p, xi, fl)
warning off Optimization:fsolve:NonSquareSystem
[Xe,FVAL,EXITFLAG,OUTPUT]=fsolve(@posexform,xi,optimset('fsolve'),p,fl);



%% plotobj.m
%  This function simply plots an object in 3-space
%  Written by LCDR Blake Eikenberry, 2005-2006

function []=plotobj(obj,p,D,def)
ind=obj.path;
path=p(:,ind);
path=path';
x=path(:,1);
y=path(:,2);
z=path(:,3);
if D==3
    plot3(x,z,y,def); hold on;
    plot3(0,0,0,'r.')
    xlabel('x'); ylabel('z'); zlabel('y');
else
    plot(x,y,def); hold on;
    plot(0,0,'r.')
    xlabel('x'); ylabel('y');;
end
axis equal; grid on;
hold off;



%% computetraj.m
%  this function computes a trajectory for the AMPHIS testbed using the
%  Direct Calculus of Variation Method
%  Written by LCDR Blake Eikenberry, 2005-2006
%  Much help from Oleg Yakimenko

function traj=computetraj(c)

% tic
% min = fminHJ(@(x) cost(x,c), [8,0,0,0,1,0,3,0,0,0,0,0,0])
% toc

min = [10.4805;0;0;0;1;0;3;.6499;.1158;.3888;.2599;-.1065;-1.1829];

[traj]=trajectory(min,c);
%[C,J,P]=cost(min,c)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [COST,J,P]=cost(FREE,CONST)

Fmax=1; Tmax=10; W=5; D=5; MSD=.41;

traj=trajectory(FREE,CONST);
time=traj.time; dtime=traj.dtime;
```

```
T=traj.T; F=traj.F; d=traj.delta;
x=traj.x; y=traj.y;
xp=traj.xp; yp=traj.yp;

% calulate cost and penalty functions

% J=[1 1 0]/2* ...
%     [time(1,end); ...
%      abs(time(2,end)-(time(1,end)))+abs(time(3,end)-time(2,end)); ...
%      sum(sum(F.*dtime))];
%
% P=[1 1 0 0 0 0 ]/2* ...
%     [sum(sum(max(0, F-Fmax).^2)); ...
%     sum(sum(max(0,abs(T)-Tmax).^2)); ...
%     sum(sum(max(0,abs(x-D/2)-D/2+MSD).^2 + max(0,abs(y-W/2)-W/2+MSD).^2));
...
%     sum(max(0,2*MSD-sqrt((xp(1,:).*dtime(1,:)-
xp(2,:).*dtime(2,:)).^2+(yp(1,:).*dtime(1,:)-yp(2,:).*dtime(2,:)).^2))); ...
%     sum(max(0,2*MSD-sqrt((xp(1,:).*dtime(1,:)-
xp(3,:).*dtime(3,:)).^2+(yp(1,:).*dtime(1,:)-yp(3,:).*dtime(3,:)).^2))); ...
%     sum(max(0,2*MSD-sqrt((xp(3,:).*dtime(3,:)-
xp(2,:).*dtime(2,:)).^2+(yp(3,:).*dtime(3,:)-yp(2,:).*dtime(2,:)).^2)))];

J=norm(time(:,end))+abs(max(max(F)));
P=norm(sum(max(0,abs(x-D/2)-D/2+MSD).^2 + max(0,abs(y-W/2)-W/2+MSD).^2));
COST=[.2 .8]*[J; P];
COST=norm(time(:,end)-[30;15;25]);
%fprintf('%0.2f, %0.2f\n', norm(time(:,end)), abs(max(max(F))))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function traj=trajectory(FREE,CONST)

TAUf=FREE(1);
% POSf=FREE(2:4)';
POSf=[3.5;2;-1];
BASE=1;
%XPPP0=FREE(BASE+(1:3)).*cos(FREE(BASE+(4:6)));
%YPPP0=FREE(BASE+(1:3)).*sin(FREE(BASE+(4:6)));
XPPP0=[0; 0; 0];
YPPP0=[0; 0; 0];

XPPPf=FREE(BASE+(7:9)).*cos(FREE(BASE+(10:12)));
YPPPf=FREE(BASE+(7:9)).*sin(FREE(BASE+(10:12)));

X0=CONST(1:3); Y0=CONST(4:6); T0=CONST(7:9);
U0=CONST(10:12); V0=CONST(13:15); W0=CONST(16:18);

PE2f=CONST(19:21); PE3f=CONST(22:24);

[IG1,IG2,STf] = pe2st(POSf, PE2f, PE3f);
Xf=STf(:,1); Yf=STf(:,2); Tf=STf(:,3);

syms tauf x0 xp0 xpp0 xppp0 xf xpf xppf xpppf real
A=[ 1 0 0 0 0 0 0 0;...
    0 1 0 0 0 0 0 0;...
    0 0 1 0 0 0 0 0;...
    0 0 0 1 0 0 0 0;...
    1  tauf tauf^2/2 tauf^3/6 tauf^4/24 tauf^5/60 tauf^6/120 tauf^7/210;...
    0  1    tauf     tauf^2/2 tauf^3/6  tauf^4/12 tauf^5/20  tauf^6/30;...
    0  0    1        tauf     tauf^2/2  tauf^3/3  tauf^4/4   tauf^5/5;...
    0  0    0        1        tauf      tauf^2    tauf^3     tauf^4];
b=[x0 xp0 xpp0 xppp0 xf xpf xppf xpppf]';
```

```
a=A\b;
a=collect(a,tauf);
N=length(a);

% define boundary conditions
% {'x0','xp0','xpp0','xppp0','xf','xpf','xppf','xpppf','tauf'}
BND{1}={X0(1),U0(1),0,XPPP0(1),Xf(1),0,0,XPPPf(1),TAUf};
BND{2}={X0(2),U0(2),0,XPPP0(2),Xf(2),0,0,XPPPf(2),TAUf};
BND{3}={X0(3),U0(3),0,XPPP0(3),Xf(3),0,0,XPPPf(3),TAUf};
BND{4}={Y0(1),V0(1),0,YPPP0(1),Yf(1),0,0,YPPPf(1),TAUf};
BND{5}={Y0(2),V0(2),0,YPPP0(2),Yf(2),0,0,YPPPf(2),TAUf};
BND{6}={Y0(3),V0(3),0,YPPP0(3),Yf(3),0,0,YPPPf(3),TAUf};
BND{7}={T0(1),W0(1),0,0,Tf(1),0,0,0,TAUf};
BND{8}={T0(2),W0(2),0,0,Tf(2),0,0,0,TAUf};
BND{9}={T0(3),W0(3),0,0,Tf(3),0,0,0,TAUf};
dtau=.5; tau=[0:dtau:TAUf];

clear A
% Calculate trajecotries (3+3 7th order case)
for i=1:9
    A{i}=subs(a,...
        {'x0','xp0','xpp0','xppp0','xf','xpf','xppf','xpppf','tauf'},...
        BND{i});
    Ax{i}     =diag([1,1,1/2,1/6,1/24,1/60,1/120,1/210])*A{i};
    Axp{i}    =diag([0,1,1,1/2,1/6,1/12,1/20,1/30])*A{i};
    Axpp{i}   =diag([0,0,1,1,1/2,1/3,1/4,1/5])*A{i};
    Axppp{i}  =diag([0,0,0,1,1,1,1,1])*A{i};
    Cx(i,:)   =Ax{i}([N:-1:1]);
    Cxp(i,:)  =Axp{i}([N:-1:2]);
    Cxpp(i,:) =Axpp{i}([N:-1:3]);
    Cxppp(i,:)=Axppp{i}([N:-1:4]);
    X(i,:)    =polyval(Cx(i,:),tau);
    Xp(i,:)   =polyval(Cxp(i,:),tau);
    Xpp(i,:)  =polyval(Cxpp(i,:),tau);
    Xppp(i,:) =polyval(Cxppp(i,:),tau);
end % i

% Put trajectories in robot terms

for i = 1:3
    x(i,:)=X(i,:);
    xp(i,:)=Xp(i,:);
    xpp(i,:)=Xpp(i,:);
    xppp(i,:)=Xppp(i,:);
    y(i,:)=X(i+3,:);
    yp(i,:)=Xp(i+3,:);
    ypp(i,:)=Xpp(i+3,:);
    yppp(i,:)=Xppp(i+3,:);
    t(i,:)=X(i+6,:);
    tp(i,:)=Xp(i+6,:);
    tpp(i,:)=Xpp(i+6,:);
    tppp(i,:)=Xppp(i+6,:);

    % Calculate Controls

    time(i,1)=1e-5; L(i,1)=1;
    u(i,1)=xp(i,1); v(i,1)=yp(i,1);
    V(i,1)=sqrt(u(i,1)^2+v(i,1)^2);
    dtime(i,1)=sqrt((x(i,2)-x(i,1))^2+(y(i,2)-y(i,1))^2)/V(i,1);
    w(i,1)=0; T(i,1)=0;

    for j = 2:length(tau)
        time(i,j)=time(i,j-1)+dtime(i,j-1); L(i,j)=dtau/dtime(i,j-1);
```

```
        u(i,j)=L(i,j)*xp(i,j); v(i,j)=L(i,j)*yp(i,j);
        V(i,j)=sqrt(u(i,j)^2+v(i,j)^2);
        dtime(i,j)=2*sqrt((x(i,j)-x(i,j-1))^2+(y(i,j)-y(i,j-
1))^2)/(V(i,j)+V(i,j-1));
        w(i,j)=(t(i,j)-t(i,j-1))/dtime(i,j-1)*2-w(i,j-1);
        T(i,j)=(w(i,j)-w(i,j-1))/dtime(i,j-1);
    end % j

    p(i,:)=atan2(ypp(i,:), xpp(i,:));
    F(i,:)=sqrt(xpp(i,:).^2+ypp(i,:).^2).*L(i,:);
    d(i,:)=((xppp(i,:).*ypp(i,:)-xpp(i,:).*yppp(i,:))./...
        (ypp(i,:)+1e-20).*cos(p(i,:)).^2-tp(i,:)).*L(i,:);
end % i

traj.time=time;
traj.T=T;
traj.F=F;
traj.delta=d;
traj.dtime=dtime;
traj.x=x;
traj.y=y;
traj.xp=xp;
traj.yp=yp;
traj.u=u;
traj.v=v;
traj.w=w;
traj.lambda=L;
traj.theta=t;
traj.psi=p;
traj.tau=tau;
traj.alpha=p-t;



%% draw_traj.m
%  this function plots trajectory information for analysis
%  Written by LCDR Blake Eikenberry, 2005-2006

function draw_traj(traj)

col='brg'; shp='-d.';

tau=traj.tau;
T=traj.T;
F=traj.F;
d=traj.delta;
dtime=traj.dtime;
x=traj.x;
y=traj.y;
L=traj.lambda;
time=traj.time;
t=traj.theta;
p=traj.psi;
u=traj.u;
v=traj.v;
w=traj.w;
alf=traj.alpha;


for i=1:3
    % plot x vs y
    figure(1)
```

```
    plot(y(i,:),x(i,:),[shp(i) col(i)]); hold on

    figure(2)
    % plot L vs tau
    subplot(5,3,1)
    plot(tau,L(i,:),[shp(i) col(i)]); hold on

    % plot time vs tau
    subplot(5,3,3)
    plot(tau,time(i,:),[shp(i) col(i)]); hold on

    % plot dtime vs tau
    subplot(5,3,2)
    plot(tau,dtime(i,:),[shp(i) col(i)]); hold on

    % plot x vs tau
    subplot(5,3,4)
    plot(tau,x(i,:),[shp(i) col(i)]); hold on

    % plot y vs tau
    subplot(5,3,5)
    plot(tau,y(i,:),[shp(i) col(i)]); hold on

    % plot theta vs tau
    subplot(5,3,6)
    plot(tau,t(i,:),[shp(i) col(i)]); hold on

    % plot psi vs tau
    subplot(5,3,15)
    plot(tau,p(i,:),[shp(i) col(i)]); hold on

    % plot F vs tau
    subplot(5,3,10)
    plot(tau,F(i,:),[shp(i) col(i)]); hold on

    % plot T vs tau
    subplot(5,3,12)
    plot(tau,T(i,:),[shp(i) col(i)]); hold on

    % plot d vs tau
    subplot(5,3,11)
    plot(tau,d(i,:),[shp(i) col(i)]); hold on

    % plot u vs tau
    subplot(5,3,7)
    plot(tau,u(i,:),[shp(i) col(i)]); hold on

    % plot v vs tau
    subplot(5,3,8)
    plot(tau,v(i,:),[shp(i) col(i)]); hold on

    % plot w vs tau
    subplot(5,3,9)
    plot(tau,w(i,:),[shp(i) col(i)]); hold on

    % plot w vs tau
    subplot(5,3,13)
    plot(y(i,:),x(i,:),[shp(i) col(i)]); hold on

    % plot alpha vs tau
    subplot(5,3,14)
    plot(tau,alf(i,:),[shp(i) col(i)]); hold on
end % i
```

```
% label plots

figure(1)
xlabel('y'), ylabel('x')
axis equal, axis([0 14 0 16]*.3048)

figure(2)
subplot(5,3,1), ylabel('\lambda')
subplot(5,3,2), ylabel('\Delta t (s)')
subplot(5,3,3), ylabel('t (s)')
subplot(5,3,4), ylabel('x (m)')
subplot(5,3,5), ylabel('y (m)')
subplot(5,3,6), ylabel('\psi (rad)')
subplot(5,3,7), ylabel('u (m/s)')
subplot(5,3,8), ylabel('v (m/s)')
subplot(5,3,9), ylabel('\omega (rad/s)')
subplot(5,3,10), ylabel('F (N/kg)')
subplot(5,3,11), ylabel('\delta (rad/s)')
subplot(5,3,12), ylabel('T (s^{-2})')
subplot(5,3,13), ylabel('TBD')
subplot(5,3,14), ylabel('\alpha (rad)')
subplot(5,3,15), ylabel('\alpha + \psi (rad)')
legend('Robot_1', 'Robot_2', 'Robot_3',0)

for i = 1:15
    subplot(5,3,i)
    xlabel('\tau'),% axis tight
end

subplot(5,3,13), ylabel('x-axis (m)'), xlabel('y-axis (m)')
axis equal, axis([0 14 0 16]*.3048)



function[x,fval,exitflag]=fminHJ(Fun,x)
%FMINHJ Multidimensional unconstrained nonlinear minimization (Hooke-Jeeves).
%   X = FMINHJ(FUN,X0) starts at X0 and attempts to find a local minimizer
%   X of the function FUN.  FUN is a function handle.  FUN accepts input X
%   and returns a scalar function value F evaluated at X. X0 can be a scalar
%   or vector.
%   [X,FVAL]= FMINHJ(...) returns the value of the objective function,
%   described in FUN, evaluated at X.
%   [X,FVAL,EXITFLAG] = FMINHJ(...) returns an EXITFLAG that describes
%   the exit condition of FMINHJ. Possible values of EXITFLAG and the
%   corresponding exit conditions are
%
%    1  FMINSEARCH converged to a solution X.
%    0  Maximum number of function evaluations or iterations reached.
%
%   Examples
%     FUN can be specified using @:
%        X = fminHJ(@sin,3)
%     finds a minimum of the SIN function near 3.
%
%   FMINHJ uses the Hooke-Jeeves pattern search (direct search) method.
%

%   Reference: Hooke, R., and Jeeves, T.A., "'Direct Search' Solution of
%   Numerical and Statistical Problems," Journal of the Assoc. Comput. Mach.,
%   Vol.8, No.2, 1961, pp.212-229.

%   Copyright 2006 by Oleg Yakimenko
```

```
Fun = fcnchk(Fun);      % place Fun into "function" (inline) form

prnt=0;                 % printout all intermediate steps

sumb=[' ','*'];

% all varied parameters should have the same scale (however here it's assumed
% that the last varied parameter is the largest, so that the step size will
% be defined using its scale)
n = length(x);          % number of varied parameters
scale = abs(x(n)); if scale == 0, scale=n; end

hvar0 = scale/10.;      % initial step size (10% of the scale)
hvarf = hvar0/1000.;    % final step size / x-tolerance (.01% of the scale)

eps = 0.000000001;      % function tolerance

k = hvar0;              % set the initial step

%% Check the original (basic) point
indexbp = 0;            % set the basic point (BP) search index
indexps = 0;            % set the pattern search (PS) index

    y = x;              % set the latest basic point
    p = x;              % set the suggested pattern search point
    b = x;              % set the previous pattern search point

fnew = feval(Fun,x);    % call minimization function
indexbp=indexbp+1;      % increment the basic point search index

fold = fnew;
ps = 0;                 % set the pattern search flag
bp = 0;                 % set the pattern search flag

    index=indexbp+indexps;
    if prnt == 1,
    varpar(index)=x(1);
    bpflag(index)=bp;
    perindex(index)=fnew;
    step(index)=k; end

  if prnt == 1
  disp(' ')
  header = ' Iteration     x       f(x)          step   BPflag   PSflag';
  disp(header)
  end

%% Keep looking for the minimum ...
%  ... while the step size k is greater than the x-tolerance and the value
%      of the objective function is greater than the function tolerance
while (k >= hvarf) & (abs(fnew) > eps)

index=indexbp+indexps;
  if prnt == 1
  disp(sprintf('%6.0f     %8.5f   %-10.3g   %6.3g      %c        %c',...
                index,     x(1),      fnew,     k,   sumb(bp+1),sumb(ps+1)));
  end

%% Continue the pattern search ...
%  ... if the objective function decreased compared to the previous 'pattern'
%      trial, continue the pattern search, i.e. make the same move in the
%      same direction
```

```
if (fold - fnew > eps) & (ps == 1)
exitflag = 'Continuing the pattern search';

   p = 2.*y-b;          % compute the suggested PS point as b+2*(y-b)
% !!! For constrained optimisation: This is the place to intervene!!!
   b = y;               % reassign the latest BP to the previous PS point
   x = p;               % assign the suggested PS point to the trial point
   y = x;

z=feval(Fun,x);         % check the lates trial point
indexps=indexps+1;      % increment the PS index

    index=indexbp+indexps;
    if prnt == 1,
    varpar(index)=x(1);
    bpflag(index)=bp;
    perindex(index)=fnew;
    step(index)=k; end

fold=fnew;
fnew=z;

%% Switch from searching around the basic point to the pattern search ...
%  ... if the objective function decrease was achieved during a search
%      around the basic point
elseif (fold - fnew > eps) & (ps == 0)
exitflag = 'Switching from BP to PS';

bp = 0;                 % lower the BP flag
ps = 1;                 % rise the PS flag

%% Stop PS, make one backward step and perform a new basic point search ...
%  ... if the last pattern step failed
elseif (fold - fnew <= eps) & (ps == 1)
exitflag = 'Stepping back to start a new BP search';

        p = b;          % set everything to be equal to the previous PS point
        y = b;
        x = b;

fnew=feval(Fun,x);
indexps=indexps+1;      % increment the PS index

    index=indexbp+indexps;
    if prnt == 1,
    varpar(index)=x(1);
    bpflag(index)=bp;
    perindex(index)=fnew;
    step(index)=k; end

fold=fnew;
ps=0;                   % lower the PS flag

%% Proceed with the search around the basic point
elseif (fold - fnew <= eps) & (ps == 0)
exitflag = 'Continuing the basic point search';

% if a search around the basic point failed, decrease the step size and
% re-examine a vicinity of the current basic point
    if bp == 1
    k=k/10.;            % decrease the step size
    end
```

```
% explore the basic point by making two steps (forward and backward) in
% each direction
            for j = 1:n
            x(j) = y(j) + k;
% !!! For constrained optimisation: This is the place to intervene!!!
            z=feval(Fun,x);
            indexbp=indexbp+1;      % increment the BP index
    index=indexbp+indexps;
    if prnt == 1,
    varpar(index)=x(1);
    bpflag(index)=1;
    perindex(index)=fnew;
    step(index)=k; end
                if z < fnew
                y(j) = x(j);
                else
                x(j) = y(j) - k;
% !!! For constrained optimisation: This is the place to intervene!!!
                z=feval(Fun,x);
                indexbp=indexbp+1;   % increment the BP index
    index=indexbp+indexps;
    if prnt == 1,
    varpar(index)=x(1);
    bpflag(index)=1;
    perindex(index)=fnew;
    step(index)=k; end
                    if z < fnew
                    y(j) = x(j);
                    else
                    x(j) = y(j);
                    end
                end
            fnew=min(z,fnew);
            end

bp = 1;                 % rise the BP flag

end                     % if end
end                     % while end

fval=fnew;

index=indexbp+indexps;
  if prnt == 1
  disp(sprintf('%6.0f     %8.5f   %-10.3g   %6.3g      %c        %c',...
                index,     x(1),      fnew,     k,  sumb(bp+1),sumb(ps+1)));
  end

if prnt == 1
close all
subplot(3,1,1,'align')
plot(varpar)
hold
plot(varpar.*bpflag,'r*')
%ylim([min(varpar) max(varpar)])
xlabel('Iteration'), ylabel('Variable Parameter')
subplot(3,1,2,'align')
plot(perindex,'r')
xlabel('Iteration'), ylabel('Performance index')
subplot(3,1,3,'align')
semilogy(step,'g')
xlabel('Iteration'), ylabel('Step size')
end
```

```
return



%% apfg
%  this function determines a point to move towards the end state
%  and avoid collisiotn

function task = guidance(gcmd, state)

task=gcmd';

v = sqrt(state(4,1)^2+state(5,1)^2);
me = state(1:2,1);
r2 = state(1:2,2);
r3 = state(1:2,3);

d1 = dist(me,task);
d2 = dist(me,r2);
d3 = dist(me,r3);

A= [cosd(90) sind(90); -sind(90) cosd(90)];

% Aviod robot2 if in the way
a= r2-me;
b= task(1:2)-me;
c= min([(a' * b)/d1/d2,1]);
ang = acos(complex(c));

% move tanget to robot
if (all([abs(ang)<.4, d2<d1]))
    task(1:2) = task(1:2) + A*a;
end

% move away from robot
if (d2<1)
    task(1:2) = task(1:2) - 2*a*v;
end

% Aviod robot3 if in the way
a= r3-me;
b= task(1:2)-me;
c= min([(a' * b)/d1/d3,1]);
ang = acos(complex(c));

% move tanget to robot
if (all([abs(ang)<.4, d3<d1]))
    task(1:2) = task(1:2) + A*a;
end

% move away from robot
if (d3<1)
    task(1:2) = task(1:2) - 2*a*v;
end

% find the distance between to robots
function d=dist(v1,v2)
d=sqrt((v1(1)-v2(1))^2+(v1(2)-v2(2))^2);
```

```
%% global_props.m
%  this function defines global properties related to robot size / shape,
%  and floor size / shape
%  Written by LCDR Blake Eikenberry, 2005-2006

function [robot, floor]=global_props
d2r=pi/180;

%% Define camera (attached to the robot's top)
robot(1).name='Blue';
robot(1).sfov=23*d2r;
robot(1).f=.1;
robot(1).ar=4/3;
robot(1).lc=[0.6 .6 1];
robot(1).dc=[0 0 1];          % Define light green and dark colors

a=1*.3048/2; b=1*.3048/2; h=.6;
robot(1).crns=[...
    -a, -b, 0;              % Robot's corners starting from the the left-
    a, -b, 0;               % bottom-floor and going clockwise (1-2-3-4);
    a,  b, 0;
    -a,  b, 0;
    -a, -b, -h;              % The same pattern is repeated at the above-floor
    a, -b, -h;               % level (5-6-7-8), i.e. 5 is located above 1, etc.
    a,  b, -h;
    -a,  b, -h];
clear a b h d2r

robot(2)=robot(1); robot(2).name='Red';
robot(2).lc=[1 .6 .6]; robot(2).dc=[1 0 0];
robot(3)=robot(1); robot(3).name='Green';
robot(3).lc=[0.6 1 .6]; robot(3).dc=[0 1 0];

floor.dim=[ 0,  0, 0;               % Square's corners starting from the origin
    16,  0, 0;               % (left bottom) and going clockwise
    16, 14, 0;
    0, 14, 0]*.3048;




%% initialize.m
% this function defines initial and final state for an AMPHIS simulation
%  Written by LCDR Blake Eikenberry, 2005-2006

% determine trajectories
SAMP_TIME=.003;
id = 3;

portConfig;

fpos.pe2 = [0.5714;  0.5236; 4.1888];
fpos.pe3 = [0.5714; -0.5236; 2.0944];

x0=[.5;3.5;2.5];
y0=[1;2;3.5];
t0=[1;2;3];
u0=[1;1;1]*1e-5;
v0=[1;1;1]*1e-5;
w0=[0;0;0];

% predetermined direct method parameters
c=[x0;y0;t0;u0;v0;w0;fpos.pe2;fpos.pe3];
```

```
TRAJ=computetraj(c);

THRUSTSIMINIT;

VAR.INITSTATE.x = x0(id);
VAR.INITSTATE.y = y0(id);
VAR.INITSTATE.t = t0(id);
VAR.INITSTATE.xd = u0(id);
VAR.INITSTATE.yd = v0(id);
VAR.INITSTATE.td = w0(id);
```

```
%% pe2st.m
%  This function takes a robots absolute postion and the pose estimation
%  of two other and calculates the range, relitave bearings, and absolute
%  postions of the entire system
%  Written by LCDR Blake Eikenberry, 2005-2006

function [rng, brg, pos] = pe2st(ref, pe2, pe3)
% this function does not compute the entire relative bearing matrix

r12 = pe2(1); b12 = pe2(2); e12 = pe2(3);
r13 = pe3(1); b13 = pe3(2); e13 = pe3(3);

r23=sqrt(r12^2+r13^2-2*r12*r13*cos(b12-b13));
b21=b12+pi-e12; b31=b13+pi-e13;

pos = [ref';
    ref'+[r12*cos(ref(3)+b12), r12*sin(ref(3)+b12), b12-b21+pi];
    ref'+[r13*cos(ref(3)+b13), r13*sin(ref(3)+b13), b13-b31+pi]];

x2=pos(2,1); y2=pos(2,2); t2=pos(2,3);
x3=pos(3,1); y3=pos(3,2); t3=pos(3,3);

b23=atan2(y3-y2, x3-x2)-t2;
b32=atan2(y2-y3, x2-x3)-t3;

rng = [0, r12, r13; r12, 0, r23; r13, r23, 0];
brg = [0, b12, b13; b21, 0, b23; b31, b32, 0];
```

```
%% portConfig.m
%  This function configures the port numbers of a robot via id#
%  Written by LCDR Blake Eikenberry, 2005-2006

PORT = 25000;

switch id
    case 1
        ndx=[1 2 3];
        PORTsab = PORT + 112;
        PORTsac = PORT + 113;
        PORTsba = PORT + 121;
        PORTsca = PORT + 131;
        PORTmab = PORT + 12;
        PORTmac = PORT + 13;
        PORTmba = PORT + 21;
        PORTmca = PORT + 31;
    case 2
```

```
        ndx=[2 1 3];
        PORTsab = PORT + 121;
        PORTsac = PORT + 123;
        PORTsba = PORT + 112;
        PORTsca = PORT + 132;
        PORTmab = PORT + 21;
        PORTmac = PORT + 23;
        PORTmba = PORT + 12;
        PORTmca = PORT + 32;
    otherwise
        ndx=[3 2 1];
        PORTsab = PORT + 132;
        PORTsac = PORT + 131;
        PORTsba = PORT + 123;
        PORTsca = PORT + 113;
        PORTmab = PORT + 32;
        PORTmac = PORT + 31;
        PORTmba = PORT + 23;
        PORTmca = PORT + 13;
end
```

```
%% THRUSTSIMINIT.m
%  This function configures physical property parameters for
%  the AMPHIS control plant
%  Written by Bill Price, 2006

PARA.Kpx=1;
PARA.Kdx=12;
PARA.Kix=0;
PARA.Kpy=PARA.Kpx;
PARA.Kdy=PARA.Kdx;
PARA.Kiy=PARA.Kix;
PARA.Kpa=1;
PARA.Kda=10;
PARA.Kia=0;
PARA.MASS=25;
PARA.Iz=.25;
PARA.MOMENTARM.x1=0;
PARA.MOMENTARM.y1=.2;
PARA.MOMENTARM.x2=PARA.MOMENTARM.x1;
PARA.MOMENTARM.y2=-PARA.MOMENTARM.y1;
PARA.h_w=.0494;
PARA.THRUSTERACCURACY=5*pi/180;
PARA.MAXTHRUSTSLEW=(500/84)*2*pi/60;
PARA.MAXTHRUSTPOS=180*pi/180;
```

```
%% anim_floor.m
%% Animate Floor
%  This function animates the AMPHIS simulation
%  Written by LCDR Blake Eikenberry, 2005-2006
%  Much help from Oleg Yakimenko

function anim_floor(state, u1, v1)
[robot_props, floor_props] = global_props;
```

```
mov = avifile('robotmov.avi','quality',100,'Compression','Indeo3','fps',5);
time=state.time;
[m,n] = size(time);
for i = 1:ceil(m/100):m
%     subplot(1,2,1);
    draw_floor(time(i));
    for j = 1 : 3
        pos=state.signals.values(1:3,j,i);
        draw_robot(pos,robot_props(j));
        switch j
            case 1
                % camera field of view
%                 draw_dev(j, pos, 'Cam', v1.signals.values(i,1));
                % 360 vectored variable thruster
%                 draw_dev(j, pos, 'Thruster', u1.signals.values(i,1:2));
                % bill's front/back -pi to pi vectored variable thruster
                draw_dev(j, pos, 'fThruster', u1.signals.values(i,1:2));
                draw_dev(j, pos, 'bThruster', u1.signals.values(i,3:4));
                % lidar output
                draw_dev(j, pos, 'Lidar', v1.signals.values(i,[1:2,4:5]));
            case 2
            case 3
        end
    end
    % draw camera
%     subplot(1,2,2);
%     data=state.signals.values(1:3,1:3,i)';
%     alf=v1.signals.values(i,1)+state.signals.values(3,1,i);
%     draw_foto(1, data, alf,0,0)

    mov = addframe(mov,getframe(gcf));
end
mov = close(mov);
```

```
%% draw_dev.m
%  This function plots various robot devices
%  Written by LCDR Blake Eikenberry, 2005-2006

function draw_dev(me, pos, type, u)
rbt_prop=global_props;
x=pos(1); y=pos(2); t=pos(3);

switch type
    case 'Cam'
        a=u+t;
        clr=rbt_prop(me).lc;
        sFoV=rbt_prop(me).sfov;
        rx=40/x; ry=40/y;
        patch(y*[1 1+ry*sin(a-sFoV) 1+ry*sin(a+sFoV) 1],...
            x*[1 1+rx*cos(a-sFoV) 1+rx*cos(a+sFoV) 1],...
            clr, 'LineStyle', 'none', 'FaceAlpha', .25);
    case 'Thruster'
        mag=u(1)*10;
        a=u(2)+t+pi;
        clr='c';
        sFoV=.05;
        rx=mag/x; ry=mag/y;
        patch(y*[1 1+ry*sin(a-sFoV) 1+ry*sin(a+sFoV) 1],...
            x*[1 1+rx*cos(a-sFoV) 1+rx*cos(a+sFoV) 1],...
```

```
            clr, 'LineStyle', 'none', 'FaceAlpha', .25);
    case 'fThruster'
        mag=u(1)*10;
        d=-rbt_prop(me).crns(1);
        a=u(2)+t;
        clr='c';
        len=.3;
        sFoV=.05;
        rx=mag; ry=mag;
        patch([y+d*sin(t) y+ry*sin(a-sFoV)+d*sin(t) y+ry*sin(a+sFoV)+d*sin(t)
y+d*sin(t)],...
            [x+d*cos(t) x+rx*cos(a-sFoV)+d*cos(t) x+rx*cos(a+sFoV)+d*cos(t)
x+d*cos(t)],...
            clr, 'LineStyle', 'none', 'FaceAlpha', .25);
        plot([y+d*sin(t), y+d*sin(t)+len*sin(a)], [x+d*cos(t),
x+d*cos(t)+len*cos(a)], 'm');
    case 'bThruster'
        mag=u(1)*10;
        d=rbt_prop(me).crns(1);
        a=u(2)+t+pi;
        clr='c';
        len=.3;
        sFoV=.05;
        rx=mag; ry=mag;
        patch([y+d*sin(t) y+ry*sin(a-sFoV)+d*sin(t) y+ry*sin(a+sFoV)+d*sin(t)
y+d*sin(t) ],...
            [x+d*cos(t) x+rx*cos(a-sFoV)+d*cos(t) x+rx*cos(a+sFoV)+d*cos(t)
x+d*cos(t)],...
            clr, 'LineStyle', 'none', 'FaceAlpha', .25);
        plot([y+d*sin(t), y+d*sin(t)+len*sin(a)], [x+d*cos(t),
x+d*cos(t)+len*cos(a)], 'm');
    case 'Lidar'
        r12 = u(1); b12 = u(2); r13 = u(3); b13 = u(4);
        x2 = x+r12*cos(t+b12); y2 = y+r12*sin(t+b12);
        x3 = x+r13*cos(t+b13); y3 = y+r13*sin(t+b13);
        plot(y2+.1*randn(1,10),x2+.1*randn(1,10),'.')
        plot(y3+.1*randn(1,10),x3+.1*randn(1,10),'.')
end

%% plot_floor.m
%  This function plots the floor for a single frame of the AMPHIS simulation
%  Written by LCDR Blake Eikenberry, 2005-2006

function draw_floor(t)
[r_props,f_props]=global_props;

% plot the floor
hold off
RedSquare=f_props.dim;
fill([RedSquare(2,3) RedSquare(3,2)], [RedSquare(1,1) RedSquare(1,2)],'w'),
hold on
axis equal, axis([RedSquare(2,2) RedSquare(3,2) RedSquare(1,1)
RedSquare(2,1)]);
title('Bird''s Eye View');
xlabel('y-axis (East) (m)'), ylabel('x-axis (North) (m)')
text('Color',[0.8471 0.1608 0],'FontAngle','italic',...
    'Position',[.1 .1],...
    'String',['time=' num2str(round(100*t)/100)])
```

```
%% draw_foto.m
%  This function plots a view of the AMPHIS floor from the perspective
%  of a robot.
%  Written by Oleg Yakimenko and Blake Eikenberry 2006

function draw_foto(ME, FLR, psy, theta, phi)
[robot, floor]=global_props;
% psy=yaw, theta=pitch, phi=roll

X=FLR(ME,1); Y=FLR(ME,2); T=FLR(ME,3);

%% Define parameters of the square in {n} (NED)
RedSquare=floor.dim;                % Square's corners starting from the
origin
NumbofPts=length(RedSquare);

%% Define camera (attached to the robot's top)
hc=robot(ME).crns(end);              % Camera's hight above the ground
Camera = [X; Y; hc];                 % Camera's position in {n}
sFoV=robot(ME).sfov;                 % Semi-field of view (horizontal)
AsRatio=robot(ME).ar;                % Frame's aspect ratio'
(horizontal/vertical)
f=robot(ME).f;                       % Focal length (m)

R_phi   = [1   0          0;
           0   cos(phi)   sin(phi);
           0  -sin(phi)   cos(phi)];
R_theta = [cos(theta)  0  -sin(theta)
           0           1   0;
           sin(theta)  0   cos(theta)];

%% Define two other robots in {b} (NED) attached to the robot's bottom

if ME==1, ROBOT1=2; ROBOT2=3; end
if ME==2, ROBOT1=1; ROBOT2=3; end
if ME==3, ROBOT1=1; ROBOT2=2; end

% Define light green and dark green colors
RobPos=[FLR(ROBOT1,1),FLR(ROBOT1,2),0;     % Origin of the robot's {b} in {n}
        FLR(ROBOT2,1),FLR(ROBOT2,2),0];
RobOr= [FLR(ROBOT1,3);FLR(ROBOT2,3)];      % Orientation of {b} wrt to {n}

L(1,:)=robot(ROBOT1).lc;
D(1,:)=robot(ROBOT1).dc;
L(2,:)=robot(ROBOT2).lc;
D(2,:)=robot(ROBOT2).dc;

CRNS{1}=robot(ROBOT1).crns;
CRNS{2}=robot(ROBOT2).crns;

NumbofRbts=length(RobOr);
for u=1:NumbofRbts
R_r2n(:,:,u)  = [cos(RobOr(u))  -sin(RobOr(u))  0;
                 sin(RobOr(u))   cos(RobOr(u))  0;
                      0               0         1];
end

%% Define a camera frame
Uscale=f*tan(sFoV);
Vscale=Uscale/AsRatio;

%% i) Convert the square to the camera frame
R_psy = [cos(psy) sin(psy) 0;
```

```
        -sin(psy) cos(psy) 0;
                0        0 1];

R_n2c = R_phi*R_theta*R_psy;                    % Rotation from {n} wrt {c}

imrs=R_n2c*(RedSquare'-Camera*ones(1,NumbofPts));% Coordinates in {c}
azimuth=atan2(imrs(2,:),imrs(1,:));

u0 = f*imrs(2,:)./imrs(1,:);                    % x-coordinate in {f} (right)
v0 =-f*imrs(3,:)./imrs(1,:);                    % y-coordinate in {f} (right)

%% ii) Count the number and indices of Visible and Invisible points
indVis=find(imrs(1,:)>0); indInv=find(imrs(1,:)<=0);
nVis=length(indVis); nInv=NumbofPts-nVis;

%% iii) Reoder the points
if (nVis~=1) & (min(indInv)>1 & max(indInv)<NumbofPts)
    fict=indVis;
    indVis=(max(indInv)+1):NumbofPts;
    indVis=[indVis 1:(min(indInv)-1)];
end

%% iv) Assign fictituous points as substitutes for invisible points
u(2:nVis+1)=u0(indVis);
v(2:nVis+1)=v0(indVis);

inleft=indVis(1)-1;     if inleft<1,          inleft=inleft+NumbofPts;   end
inright=indVis(nVis)+1; if inright>NumbofPts, inright=inright-NumbofPts; end

tau1=abs((-sFoV-azimuth(indVis(1)))/(azimuth(inleft)-azimuth(indVis(1))));
tau2=abs((sFoV-azimuth(indVis(nVis)))/(azimuth(inright)-
azimuth(indVis(nVis))));
imrLeft=imrs(:,inleft)*tau1+imrs(:,indVis(1))*(1-tau1);
imrRight=imrs(:,inright)*tau2+imrs(:,indVis(nVis))*(1-tau2);

ul = f*imrLeft(2)/imrLeft(1);       % Coordinates of fictituous points in {f}
vl =-f*imrLeft(3)/imrLeft(1);
ur = f*imrRight(2)/imrRight(1);
vr =-f*imrRight(3)/imrRight(1);

u(1)=(-Vscale-vl)*(u(2)-ul)/(v(2)-vl)+ul; v(1)=-Vscale;
u(nVis+2)=(-Vscale-vr)*(u(nVis+1)-ur)/(v(nVis+1)-vr)+ur; v(nVis+2)=-Vscale;

%% v) Convert robots centers from {n} to {c}
imRts=R_n2c*(RobPos'-Camera*ones(1,NumbofRbts)); % Robots coordinates in {c}
distRts=[norm(imRts(:,1),2) norm(imRts(:,2),2)]; % Distance from the origin of
{c}
azimuthRts=atan2(imRts(2,:),imRts(1,:));        % Robots azimuths in {c}

for jr=1:NumbofRbts
    % vi) Convert robot's corners from {b} to {n}
    Robot=CRNS{jr};
    cyl=sqrt(Robot(1)^2+Robot(2)^2);          % Cylinder around the robot
    RobCrns(:,:,jr)=R_r2n(:,:,jr)*Robot'+RobPos(jr,:)'*ones(1,8);
    % vii) Default zeroing of left and right planes' coordinates (in {f})
    uR(:,2*jr-1) = zeros(4,1);  vR(:,2*jr-1) = zeros(4,1);
    uR(:,2*jr)   = zeros(4,1);  vR(:,2*jr)   = zeros(4,1);
    uRcolor(:,:,jr)=[L(jr,:);D(jr,:)];

    if abs(azimuthRts(jr))-sFoV<pi/2 && distRts(jr)*sin(abs(azimuthRts(jr))-
sFoV)<cyl
        %% viii) Convert visible robot's corners from {n} to {c}
```

```
        imRtsCrns(:,:,jr)=R_n2c*(RobCrns(:,:,jr)-Camera*ones(1,8));   %
Coordinates in {c}
        %% ix) Determine the closest edge and two adjacent panels (left and
right)
        [dv,in]=min([norm(imRtsCrns(1:3,1,jr)),norm(imRtsCrns(1:3,2,jr)),...
            norm(imRtsCrns(1:3,3,jr)),norm(imRtsCrns(1:3,4,jr))]);

        inL=in+1;       if inL>4,   inL=inL-4;    end
        inR=in-1;       if inR<1,   inR=inR+4;    end

        Panel(:,:,2*jr-1)=[imRtsCrns(:,in,jr),imRtsCrns(:,inL,jr),...   % Left
panel
                           imRtsCrns(:,inL+4,jr),imRtsCrns(:,in+4,jr)];
        Panel(:,:,2*jr)  =[imRtsCrns(:,in,jr),imRtsCrns(:,inR,jr),...   % Right
panel
                           imRtsCrns(:,inR+4,jr),imRtsCrns(:,in+4,jr)];

        %% x) Determine more distant panel (left or right) to be shown first
        dl=norm(mean(Panel(:,:,2*jr-1),2));
        dr=norm(mean(Panel(:,:,2*jr),2));
        tt=[0,1];
        if dl<dr, tt=[1,0]; uRcolor(:,:,jr)=[D(jr,:);L(jr,:)]; end

        %% xi) Compute {f}-coordinates of the farther and closer panels
        for jt=1:2
            uR(:,2*jr-1+tt(jt))= f*Panel(2,:,2*jr-2+jt)./Panel(1,:,2*jr-2+jt);
            vR(:,2*jr-1+tt(jt))=-f*Panel(3,:,2*jr-2+jt)./Panel(1,:,2*jr-2+jt);
        end

    end             % The end of the 'if' structure
end                 % The end of the 'for' loop

ord=[2,1]; if distRts(2)<distRts(1), ord=[1,2]; end

%     u(5)=u(1); v(5)=v(1);
%     uR(5,:)=uR(1,:); vR(5,:)=vR(1,:);

fill([-1 1 1 -1], [-1 -1 1 1], 'w','FaceAlpha', 1)
patch(u,v,'c','FaceAlpha', 1);
patch(uR(:,2*ord(1)-1),vR(:,2*ord(1)-1),uRcolor(1,:,ord(1)), 'FaceAlpha', 1);
patch(uR(:,2*ord(1)),  vR(:,2*ord(1)),  uRcolor(2,:,ord(1)), 'FaceAlpha', 1);
patch(uR(:,2*ord(2)-1),vR(:,2*ord(2)-1),uRcolor(1,:,ord(2)), 'FaceAlpha', 1);
patch(uR(:,2*ord(2)),  vR(:,2*ord(2)),  uRcolor(2,:,ord(2)), 'FaceAlpha', 1);
axis equal, axis([-Uscale Uscale -Vscale Vscale]);
title (['SimImage from ' robot(ME).name])



%% draw_robot.m
%  this function draws a single robot on the AMPHIS floor
%  Written by LCDR Blake Eikenberry, 2005-2006

function draw_robot(pos,robot)
x=pos(1); y=pos(2); t=pos(3); % postion/orientation of robot

% Convert robot's corners from {b} to {n}
r2n  = [cos(t) -sin(t)  0;
        sin(t)  cos(t)  0;
           0        0   1];
RobCrns=r2n*robot.crns'+[x;y;0]*ones(1,8);
fill(RobCrns(2,1:4,1),RobCrns(1,1:4,1),robot.dc)
radius=abs(robot.crns(1));
line([y y+radius*sin(t)],[x x+radius*cos(t)], 'Color', 'y')
```

```
%% closeport.m
%  This function closes and deletes a serial port object
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function closeport(port)

if port~=0
    fclose(port)
    delete(port)
end



%% hex2decword.m
%  This function takes a vector of hex ascii values, pairs them up,
%  and converts each pair to a decimal value
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function out = hex2decword(in)
try
out=[]; j=1;
if any([mod(length(in),4), isempty(in)]), error('wrong size'), end
for i = 1:4:length(in)
    out(j)=hex2dec(char(in(i:i+3)));
    j=j+1;
end
catch
    char (in)
    error('CRASH in hex2decword')
end



%% LidarBaud.m
%  This function sets the Baud rate on the SICK LIDAR
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

disp('Set Baud Rate')
data=double('00100008000002010001000600000010008');
writedata(s,data,PRNT);
buffer=readdata(s);
[buffer,msg]=LidarParse(buffer,PRNT);

data=double('00100004000002020001');
writedata(s,data,PRNT);
buffer=readdata(s);
[buffer,msg]=LidarParse(buffer,PRNT);



%% LidarCRC.m
%  This function calculates the CRC for the SICK LIDAR
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function crc=LidarCRC(data)
% Abstract:
% routines for calculating a 16 bits CRC signature using the generator
% polynom x^16 + x^12 + x^5 + 1 as recommended by the ITU.T V.42
% (former CCITT); all routines use a table driven algorithm
% XOR table for CRC algorithm, CRC-16, ITU.T X.25
% polynomial: h1021
crctab = {
    '0000', '1021', '2042', '3063', '4084', '50a5', '60c6', '70e7',...
```

```
    '8108', '9129', 'a14a', 'b16b', 'c18c', 'd1ad', 'e1ce', 'f1ef',...
    '1231', '0210', '3273', '2252', '52b5', '4294', '72f7', '62d6',...
    '9339', '8318', 'b37b', 'a35a', 'd3bd', 'c39c', 'f3ff', 'e3de',...
    '2462', '3443', '0420', '1401', '64e6', '74c7', '44a4', '5485',...
    'a56a', 'b54b', '8528', '9509', 'e5ee', 'f5cf', 'c5ac', 'd58d',...
    '3653', '2672', '1611', '0630', '76d7', '66f6', '5695', '46b4',...
    'b75b', 'a77a', '9719', '8738', 'f7df', 'e7fe', 'd79d', 'c7bc',...
    '48c4', '58e5', '6886', '78a7', '0840', '1861', '2802', '3823',...
    'c9cc', 'd9ed', 'e98e', 'f9af', '8948', '9969', 'a90a', 'b92b',...
    '5af5', '4ad4', '7ab7', '6a96', '1a71', '0a50', '3a33', '2a12',...
    'dbfd', 'cbdc', 'fbbf', 'eb9e', '9b79', '8b58', 'bb3b', 'ab1a',...
    '6ca6', '7c87', '4ce4', '5cc5', '2c22', '3c03', '0c60', '1c41',...
    'edae', 'fd8f', 'cdec', 'ddcd', 'ad2a', 'bd0b', '8d68', '9d49',...
    '7e97', '6eb6', '5ed5', '4ef4', '3e13', '2e32', '1e51', '0e70',...
    'ff9f', 'efbe', 'dfdd', 'cffc', 'bf1b', 'af3a', '9f59', '8f78',...
    '9188', '81a9', 'b1ca', 'a1eb', 'd10c', 'c12d', 'f14e', 'e16f',...
    '1080', '00a1', '30c2', '20e3', '5004', '4025', '7046', '6067',...
    '83b9', '9398', 'a3fb', 'b3da', 'c33d', 'd31c', 'e37f', 'f35e',...
    '02b1', '1290', '22f3', '32d2', '4235', '5214', '6277', '7256',...
    'b5ea', 'a5cb', '95a8', '8589', 'f56e', 'e54f', 'd52c', 'c50d',...
    '34e2', '24c3', '14a0', '0481', '7466', '6447', '5424', '4405',...
    'a7db', 'b7fa', '8799', '97b8', 'e75f', 'f77e', 'c71d', 'd73c',...
    '26d3', '36f2', '0691', '16b0', '6657', '7676', '4615', '5634',...
    'd94c', 'c96d', 'f90e', 'e92f', '99c8', '89e9', 'b98a', 'a9ab',...
    '5844', '4865', '7806', '6827', '18c0', '08e1', '3882', '28a3',...
    'cb7d', 'db5c', 'eb3f', 'fb1e', '8bf9', '9bd8', 'abbb', 'bb9a',...
    '4a75', '5a54', '6a37', '7a16', '0af1', '1ad0', '2ab3', '3a92',...
    'fd2e', 'ed0f', 'dd6c', 'cd4d', 'bdaa', 'ad8b', '9de8', '8dc9',...
    '7c26', '6c07', '5c64', '4c45', '3ca2', '2c83', '1ce0', '0cc1',...
    'ef1f', 'ff3e', 'cf5d', 'df7c', 'af9b', 'bfba', '8fd9', '9ff8',...
    '6e17', '7e36', '4e55', '5e74', '2e93', '3eb2', '0ed1', '1ef0'};

numofbytes = length(data)/4;
initial_crc = 'ffff';

i=1;
crc=uint16(hex2dec(initial_crc));
while(numofbytes)
    numofbytes = numofbytes-1;
    d = uint16(hex2dec(char(data(i:i+3)))); i = i+4;

    a1=bitshift(crc,8);
    a2=bitand(bitshift(d,-8),255);
    nd=bitshift(crc,-8);
    a3=hex2dec(crctab(nd+1));
    a4=bitor(a1,a2);
    a5=bitxor(a4,a3);

    crc=a5;

    a1=bitshift(crc,8);
    a2=bitand(d,255);
    nd=bitshift(crc,-8);
    a3=hex2dec(crctab(nd+1));
    a4=bitor(a1,a2);
    a5=bitxor(a4,a3);

    crc = a5;
end

crc=double(dec2hex(crc));

while length(crc)<4  % pad with ZERO if required
```

```
    crc = [48 crc];
end



%% LidarGetprofile
%  This function queries the SICK LIDAR for a profile
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function profile = LidarGetprofile(s,PRNT);

% global s

buffer=[]; profile = [];
data=double('00100005000003010001018B0'); % CRC = '13DD'
writedata(s,data,PRNT);

while length(buffer) ~= 2430
    new=readdata(s);
    if isempty(new),
        return,
    end
    buffer=[buffer new];
end

[buffer,seg]=LidarParse(buffer,PRNT);
seg = hex2decword(seg);

ffff = seg(1);
numofsegs = seg(2);
profile = seg(3:end); % remove ffff and segment number

for i = numofsegs-1:-1:1
    [buffer,seg]=LidarParse(buffer,PRNT);
    seg = hex2decword(seg);
    seg(1)=[];  % remove segment number
    profile=[profile, seg];
end



%% LidarIdle.m
%  This function makes the SICK LIDAR idle
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

disp('Idle')
data=double('0010000300000402');  % CRC = '6836'
writedata(s,data,PRNT);

buffer=readdata(s);
[buffer,msg]=LidarParse(buffer, PRNT);



%% LidarMeasure.m
%  This function makes the rotating SICK LIDAR activate range measuring
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

disp('Measure')
data=double('0010000300000404'); % CRC='6830'
writedata(s,data,PRNT);

buffer=readdata(s);
```

```
[buffer,msg]=LidarParse(buffer, PRNT);



%% LidarMeasureStop.m
%  This function stops the SICK LIDAR from range measuring
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

disp('Spin')
data=double('00100004000004030000'); % CRC = 'A3E3'
writedata(s,data,PRNT);
buffer=readdata(s);
[buffer,msg]=LidarParse(buffer, PRNT);



%% LidarParse.m
%  This function parses data from SICK LIDAR
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function [res,cmd]=LidarParse(str, PRNT)
res = str; cmd = [];

if isempty(str), return, end

m=str(find(str==2,1):find(str==3,1));
if length(m)>0
    res = str(1+find(str==3,1) : end );

    STX=m(1);
    SID=char(m(2:3));
    DID=char(m(4:5));
    LEN=hex2dec(char(m(6:9)));
    ETX=m(end);

    CRC=m(end-4:end-1);
    crc=LidarCRC(m(2:end-5));
    cmd = m(10:end-5);

    if PRNT
        fprintf('STX = %2i SID = %2s DID = %2s LEN = %5i ETX =
%2i\n',STX,SID,DID,LEN,ETX)
        fprintf('Lidar Message: ')
        fprintf('%c', cmd)
        if hex2dec(char(CRC)) == hex2dec(char(crc))
            fprintf(': CRC match = %c%c%c%c\n', CRC)
        else
            fprintf(': CRC ERR %c%c%c%c//%c%c%c%c\n', crc, CRC)
            error('STOPPING')
        end
    end
end



%% LidarProfile.m
%  This function creates a profile from the data buffer from the SICK LIDAR
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function [profileout,bufferout] = LidarProfile(bufferin,PRNT)
profileout=[]; bufferout=bufferin;
[buffer,seg]=LidarParse(bufferin,PRNT);
if ~isempty(seg)
```

```
    seg = hex2decword(seg);
else
    return
end

ffff = seg(1);
numofsegs = seg(2);
profile = seg(3:end); % remove ffff and segment number

for i = numofsegs-1:-1:1
    [buffer,seg]=LidarParse(buffer,PRNT);
    if ~isempty(seg)
        seg = hex2decword(seg);
    else
        return
    end
    seg(1)=[];  % remove segment number
    profile=[profile, seg];
end

profileout=profile;
bufferout=buffer;
```

```
%% LidarRead.m
%  This function reads datas from the SICK LIDAR
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function [profile,buffer] = LidarRead(s,bufferin,PRNT)

while length(buffer) ~= 2430
    new=readdata(s);
    if isempty(new),
        return,
    end
    buffer=[buffer new];
end

[buffer,seg]=LidarParse(buffer,PRNT);
seg = hex2decword(seg);

ffff = seg(1);
numofsegs = seg(2);
profile = seg(3:end); % remove ffff and segment number

for i = numofsegs-1:-1:1
    [buffer,seg]=LidarParse(buffer,PRNT);
    seg = hex2decword(seg);
    seg(1)=[];  % remove segment number
    profile=[profile, seg];
end
```

```
%% LidarSpin.m
%  This function starts the SICK LIDAR spinning
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

disp('Spin')
data=double('00100004000004030000'); % CRC = 'A3E3'
writedata(s,data,PRNT);
```

```
pause(8)
buffer=readdata(s);
[buffer,msg]=LidarParse(buffer, PRNT);



%% LidarTest.m
%  This script tests the SICK LIDAR
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

if exist('s'), closeport(s), end
format compact, close all, clear all; clc
s=0; PROERR=0; buffer=[]; PRNT=0; PLT=0;
%% Lidar Initialize;
s = openport('COM1',115200)   %opens port with correct settings
LidarBaud; LidarSpin; LidarMeasure;
%% Use Lidar
a=clock;
aviobj = avifile('lidarex.avi')
for cnt = 1:111
    profile = LidarGetprofile(s,PRNT);
    if isempty(profile)
        PROERR=PROERR+1;
    else
        GPS = [.5 .5 pi/180*209];
        clf
        bra = poseProfile(profile,GPS);
        brasize = length(bra);
        fprintf('1: %0.2f, %0.2f ; 2: %0.2f, %0.2f\n', ...
            bra(1), bra(2), bra(4), bra(5))
        for i = 1:3:brasize-1
            if bra(i:i+1)>0
                x=GPS(1)+bra(i+1)*cos(bra(i)+GPS(3));
                y=GPS(2)+bra(i+1)*sin(bra(i)+GPS(3));
                % DEBUG - Plot the robots on the floor
                plot(y,x,'rs'), hold on
            end
        end
        %pause(.35)
        frame = getframe(gca);
        aviobj = addframe(aviobj,frame);
    end
end
aviobj = close(aviobj);

b=clock;
fprintf('%2.2f secs\n', b(4)*60+b(5)*60+b(6)-a(4)*60-a(5)*60-a(6))
%% Quit
LidarMeasureStop;
LidarIdle;
closeport(s)
clear s
PROERR



%% openport.m
%  This function creates and opens a serial port object
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function s = openport(port,baudrate)
s = serial(port, 'BaudRate', baudrate);
fopen(s);
```

```
%% plotProfile.m
%  This function plots a single profile from the SICK LIDAR
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function [xy] = plotProfile(profile, PRNT)
if length(profile)<5, return, end
if PRNT
    fprintf('LD response %04x \n', profile(1));
    fprintf('PROFILEFORMAT %04x \n', profile(2));
    fprintf('PROFILEINFO %04x \n', profile(3));
    fprintf('SEC1: Angle step %0.3f deg \n', profile(4)/16)
    fprintf('SEC1: Number of points of sector %i \n', profile(5))
    fprintf('SEC1: Start direction of %0.3f deg \n', profile(6)/16)
    fprintf('SEC2: Angle step %0.3f deg \n', profile(7)/16)
    fprintf('SEC2: Number of points of sector %i \n', profile(8))
    fprintf('SEC2: Start direction of %0.3f deg \n', profile(9)/16)
end

deltheta = profile(7)/16*pi/180;
numpoints = profile(8);
startdir = profile(9)/16*pi/180;
point = profile(10:end)/256;
numpoints = length(point);

for i = 1:numpoints
    theta = startdir + (i-1) * deltheta;
    y(i) = point(i)*sin(theta);
    x(i) = point(i)*cos(theta);
end

xy = [x' y'];

% clf
% plot(0,0,'rs'), hold on
clf
plot(xy(:,2),xy(:,1), 'y.')
hold on
axis equal
pause(.05)



%% poseProfile.m
% This function takes the postion of the lidar (robot) in absolute
% coordinates, and the profile returned from the lidar, and returns
% the pose - a vector (1x7) of bearings, ranges, and angles + time
% of the relative postion of the other 2 robots.
%
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function bra = poseProfile(profile, pos)
% Allocate output variable (max = 10 robots)
bra = zeros(1,10*3+1);

% Check for good Profile
if length(profile)<5, return, end

% Poition of the lidar (abs) and the floor
xr=pos(1); yr=pos(2); tr=pos(3);
crns = [0,0; 0,14; 16,14; 16,0; 0,0] *.3048;

% Rotation matrix to operate on the profile
```

```
rot = [cos(tr) sin(tr); -sin(tr) cos(tr)];

% Lidar profile data
deltheta = profile(7)/16*pi/180;
numpoints = profile(8);
startdir = profile(9)/16*pi/180;
point = profile(10:end)/256;
numpoints = length(point);

% Process lidar data - group points to make objects and
% xfer relative frame into the abs frame
i=1; obj=1;
b=zeros(1,numpoints); r=zeros(1,numpoints); o=zeros(1,numpoints);
for j = 1:numpoints
    theta = startdir + (i-1) * deltheta;
    b(i)= theta;
    r(i)= point(i);
    if i > 1
        % if contiguous points are far apart, associate with new object
        if abs(r(i) - r(i-1)) > .2
            obj = obj + 1;
        end
    end
    o(i) = obj;
    X = [point(i)*cos(theta), point(i)*sin(theta)] * rot;
    x(i) = X(1) + xr;
    y(i) = X(2) + yr;
    i=i+1;
end

xy = [x' y'];
bro = [b' r' o'];

% Connect points if broken at 0/360 degress
if abs(bro(1,3)-bro(end,3)) < .2
    n = find(bro(:,3)==max(bro(:,3)));
    bro(n,1) = bro(n,1)-2*pi;
    bro(n,3) = 1;
end

% Filter out tiny objects
er=[]; j =1;
for i = 1:max(bro(:,3))
    n=find(bro(:,3)==i);
    rp = xy(min(n),:);
    lp = xy(max(n),:);
    leng=sqrt((rp(1)-lp(1))^2+(rp(2)-lp(2))^2);
    if leng < .2
        er=[er; n];
    else
        bro(n,3) = j;
        j=j+1;
    end
end
xy(er,:)=[];
bro(er,:)=[];

% Filter out objects not on the floor and find a bearing, range and
% attitiude for each object
j=1;
for i = 1 : max(bro(:,3))
    n=find(bro(:,3)==i);
    bear = abs(bro(max(n),1)+bro(min(n),1))/2;
```

```
    dist = min(bro(n,2))+.3/2;

    % DEBUG - plot the objects w/ numbers
    % fprintf('%2i %5.2f %5.2f\n', i, bear, dist)
    text(xy(min(n),2), xy(min(n),1)-.3, sprintf('%i',i)), hold on
    if mod(bro(max(n),3),2) clr = 'g'; else clr = 'b'; end
    plot(xy(n,2),xy(n,1), [clr '.']), hold on

    % Filter objects if any point is off the floor
    if any([any([xy(n,2)<0]),any([xy(n,1)<0]), ...
            any([xy(n,1)>crns(3,1)]),any([xy(n,2)>crns(3,2)])])
        % OFF_FLOOR=xy(n,:)
    else
        bra(j:j+2) = [bear, dist, 0];
        j=j+3;
    end
end

% DEBUG - Plot the floor and lidar position/orientation
plot(yr,xr,'rs'), hold on
plot([yr yr+.5*sin(tr)],[xr xr+.5*cos(tr)],'r')
plot(crns(:,2), crns(:,1), 'k')
axis equal

% Format the output variable
bra(7)=-1;
bra=bra(1:7);



%% readdata.m
%  This function reads data from the serial port for the SICK LIDAR
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function out = readdata(port)
tic; flag = 1; buffer = [];

while flag
    if port~=0,
        while port.BytesAvailable > 0
            in = fread(port,port.BytesAvailable);
            buffer = [buffer, in'];
        end
    end
    if toc > 1, flag = 0; disp('Lidar timed out during read'), end
    if ~isempty(buffer)
        if buffer(end)==3, flag = 0; end
    end
end

out=double(buffer);



%% writedata.m
%  This function writes data to the serial port for the SICK LIDAR
%  Written by LCDR Blake Eikeberry, NPS, 2005-2006

function err = writedata(port,data, PRNT)
% data is a 1xN array of double

if mod(length(data),4)
```

```
    err = 1;
else
    err = 0;
    CRC=LidarCRC(data);
    if PRNT
        fprintf('SEND:'),
        fprintf('%c', data),
        fprintf(' %c%c%c%c\n', CRC)
    end
    if port ~= 0,
        try
            fwrite(port,[2 data CRC 3])
        catch
            disp('Lidar timeout during write')
        end
    end
end
```

```
%% CW.m
%% CW/Hill's Equation Real-time Interactive Simulator
%
% MA4362 Advanced Astrodynamics / Prof. Donald Danielson
% Code written by LCDR Blake Eikenberry, Spring 2006
%  for Relative Motion and Proximity Operations Project with
%  LCDR Jason Hall, LT Bill Price, and LT Ryan Lewis
%
% References:
% Vallado, David A., Fundamentals of Astrodynamics and Applications, Microcosm
Press, 2001
% Alfriend, Terry, Notes on Relative Motion of Neighboring Satellites, NPS,
2006
% Newman, Jim, Lectures on RPOP and the Space Shuttle/ISS rendezvous, NPS, 2006
% >> Help screen available by pressing '?' <<

clc; close all; format compact; clear all;
global code, code = 100;
a=7e3; Mu=398600.4415;
n=sqrt(Mu/a^3);

fuel=0; tic; t0=clock; trail=0; az=45; el=15; inc=0;
pdtr = 1; cntr = 1; pdtrl= round(2*pi/n); zoom = 50; dV=.1; scrnmode=1;
figure('KeyPressFcn', @userevent);
disp('Press ? for help')
% aviobj = avifile('CWmovie.avi')

% Set initial conditions
ic = 1;
switch ic
    case 1 % Specified
        x0=100;y0=120;z0=140;
        u0=0; v0=0; w0=0;
    case 2 % Circle on relative orbit plane
        C=6; D2=3*C^2;
        x0=0; y0=2*C; z0=0;
        u0=n*C; v0=0; w0=n*sqrt(D2-z0^2);
    case 3 % Circle on projected y-z plane
        C=6; D2=4*C^2;
        x0=0; y0=2*C; z0=0;
        u0=n*C; v0=0; w0=n*sqrt(D2-z0^2);
```

```
    case 4 % Lagging y
        C=6; D2=3*C^2;
        xc=1; yc0=1; zc=1; vc=3*n*xc/-2;
        x0=xc; y0=2*C+yc0; z0=zc;
        u0=n/2*(y0-yc0);
        v0=(vc+6*n*x0)/-3;
        w0=n*sqrt(D2-z0^2);
    otherwise % Random
        x0=10*randn(1); y0=10*randn(1); z0=10*randn(1);
        u0=randn(1); v0=randn(1); w0=randn(1);
end

while code
    tt0=toc; inc=inc+1;
    t = linspace(tt0, tt0+pdtrl, 1000);
    psy = n*t;
    x = -(2*v0/n+3*x0)*cos(psy) + (u0/n)*sin(psy)+(4*x0+2*v0/n);
    y = (y0-2*u0/n)+(4*v0/n+6*x0)*sin(psy)+2*u0/n*cos(psy)-(6*x0+3*v0/n)*psy;
    z = z0*cos(psy)+(w0/n)*sin(psy);
    u = (2*v0+3*x0*n)*sin(psy) + (u0)*cos(psy);
    v = (4*v0+6*x0*n)*cos(psy) - 2*u0*sin(psy)-(6*x0*n+3*v0);
    w = -z0*sin(psy)*n+w0*cos(psy);
    d = norm([x(1); y(1); z(1)]);       % cartesian distance
    tti=findmindV([x(1),y(1),z(1),n]);  % velocity optimal time-to-intercept
    vo=((6*x(1)*(n*tti-sin(n*tti))-y(1))*n*sin(n*tti)-2*n*x(1)*(4-
3*cos(n*tti))*(1-cos(n*tti)))/...
        ((4*sin(n*tti)-3*n*tti)*sin(n*tti)+4*(1-cos(n*tti))^2);
    uo=-(n*x(1)*(4-3*cos(n*tti))+2*(1-cos(n*tti))*vo)/sin(n*tti);
    wo=-z(1)*n*cot(n*tti);
    ve=-2*x(1)*n;
    du=0; dv=0; dw=0;

    % Trail for View 5
    Psy=n*etime(clock,t0);
    R=[cos(Psy),-sin(Psy),0;sin(Psy),cos(Psy),0;0,0,1];
    trail=trail+1;
    P(:,trail)=R*[x(1);y(1);z(1)];
    Pt(:,trail)=R*[x(1);y(1);z(1)]+a*R(:,1);

    % Act on user interaction
    switch code
        case 1, du=-dV; dv=-dV;
        case 2, du=-dV;
        case 3, du=-dV; dv=dV;
        case 4, dv=-dV;
        case 5, du=dV*u(1);dv=dV*v(1);dw=dV*w(1);
        case 6, dv=dV;
        case 7, du=dV;  dv=-dV;
        case 8, du=dV;
        case 9, du=dV;  dv=dV;
        case 10, du=-u(1); dv=-v(1); dw=-w(1);
        case 11, du=uo-u(1); dv=vo-v(1); dw=wo-w(1);
        case 12, dw=dV;
        case 13, dw=-dV;
        case 14, dv=ve-v(1); pdtrl= round(2*pi/n);
        case 100, zoom=1+.12*d;
        case 101, zoom=zoom/1.5;
        case 102, zoom=zoom*1.5;
        case 103, pdtr=pdtr*-1;
        case 104, cntr=cntr*-1;
        case 105, pdtrl=round(pdtrl*1.3);
        case 106, pdtrl=max(round(pdtrl/1.3),10);
        case 107, dV=dV+.01;
```

```
        case 108, dV=max(dV-.01,.01);
        case 200, scrnmode=0;
        case 201, scrnmode=1;
        case 202, scrnmode=2;
        case 203, scrnmode=3;
        case 209, scrnmode=4;
        case 210, scrnmode=5;
        case 204, az=az+5;
        case 205, az=az-5;
        case 206, el=el+5;
        case 207, el=el-5;
        case 208, pause(5)
    end
    if code >0 && code < 100
        x0=x(1); y0=y(1); z0=z(1);
        v0=v(1)+dv; u0=u(1)+du; w0=w(1)+dw;
        tic; fuel=norm([du;dv;dw])+fuel;
        fprintf('dV = [%6.2f, %6.2f, %6.2f] {%5.2f} m/s\n', du, dv, dw, fuel)
    end
    if code, code =-1; end
    plotorb
end
close all
% aviobj = close(aviobj);



%% userevent.m
%% Change the action 'code' based on keys pressed by the user
%  Code written by LCDR Blake Eikenberry, Spring 2006

function [code]=userevent(src,evnt)
global code
switch evnt.Character
    case 'q', code = 0;
    case '1', code = 1;
    case '2', code = 2;
    case '3', code = 3;
    case '4', code = 4;
    case '5', code = 5;
    case '6', code = 6;
    case '7', code = 7;
    case '8', code = 8;
    case '9', code = 9;
    case '0', code = 10;
    case '.', code = 11;
    case '+', code = 12;
    case '-', code = 13;
    case '*', code = 14;
    case ',', code = 100;
    case '>', code = 101;
    case '<', code = 102;
    case 'p', code = 103;
    case 'c', code = 104;
    case ']', code = 105;
    case '[', code = 106;
    case '}', code = 107;
    case '{', code = 108;
    case '?', code = 200;
    case '!', code = 201;
    case '@', code = 202;
    case '#', code = 203;
    case 'g', code = 204;
```

```
    case 'h', code = 205;
    case 'b', code = 206;
    case 'y', code = 207;
    case ')', code = 98;
    case '(', code = 99;
    case '/', code = 208;
    case '$', code = 209;
    case '%', code = 210;
end



%% findminV.m
%  this function finds the velocity optimal time-to-intercept
%  Written by LCDR Blake Eikenberry, 2005-2006

function mint=findmindV(IN)
mint=fminsearch(@(x) dV(IN,x), 1);

function V=dV(IN,tti)
n=IN(4);
x=IN(1); y=IN(2); z=IN(3);
vo=((6*x(1)*(n*tti-sin(n*tti))-y(1))*n*sin(n*tti)-2*n*x(1)*(4-3*cos(n*tti))*(1-
cos(n*tti)))/...
    ((4*sin(n*tti)-3*n*tti)*sin(n*tti)+4*(1-cos(n*tti))^2);
uo=-(n*x(1)*(4-3*cos(n*tti))+2*(1-cos(n*tti))*vo)/sin(n*tti);
wo=-z(1)*n*cot(n*tti);
V=sqrt(vo^2+uo^2+wo^2);


%% plotorb.m
%% Plot the orbit based on the current screen mode
%  Code written by LCDR Blake Eikenberry, Spring 2006

hold off
scrnsize=[-16 16 -10 10]*zoom;

switch scrnmode
    case 1 % XY Plane
        plot(y(1),x(1),'cs'), hold on;
        if pdtr>0
            plot (y,x, 'y:')
            text(y(end), x(end), sprintf('%0.1f min', pdtrl/60), 'Color',
'y','FontSize', 8)
            text(scrnsize(1)*.99, scrnsize(3)*.9, ['v_e' sprintf('=%6.2f
m/s',ve)], 'Color', 'y','FontName', 'Courier')
            text(scrnsize(2)*.5, scrnsize(3)*.8, ['u_i' sprintf('=%6.2f
m/s',uo)], 'Color', 'y','FontName', 'Courier')
            text(scrnsize(2)*.5, scrnsize(3)*.9, ['v_i' sprintf('=%6.2f
m/s',vo)], 'Color', 'y','FontName', 'Courier')
            text(scrnsize(2)*.5, scrnsize(3)*.7, ['t_i' sprintf('=%6.1f
min',tti/60)], 'Color', 'y','FontName', 'Courier')
        end
        if cntr>0, plot([0 0],[0 zoom], 'g'), plot([0 zoom],[0 0],'g',
'LineWidth', 2),plot([0 0],[0 -zoom],'g:'), end
        text(scrnsize(2)*.5, scrnsize(4)*.9, sprintf('x =%6.2f m',x(1)),
'Color', 'g','FontName', 'Courier')
        text(scrnsize(2)*.5, scrnsize(4)*.8, sprintf('y =%6.2f m',y(1)),
'Color', 'g','FontName', 'Courier')
        text(scrnsize(2)*.5, scrnsize(4)*.7, sprintf('u =%6.2f m/s',u(1)),
'Color', 'g','FontName', 'Courier')
        text(scrnsize(2)*.5, scrnsize(4)*.6, sprintf('v =%6.2f m/s',v(1)),
'Color', 'g','FontName', 'Courier')
```

```
        xlabel('y-axis (m)'), ylabel('IN PLANE x-axis (m)')
    case 2 % ZY Plane
        plot(y(1),z(1),'cs'), hold on;
        if pdtr>0
            plot(y,z,'y:')
            text(y(end), z(end), sprintf('%0.1f min',pdtrl/60), 'Color',
'y','FontName', 'Courier')
            text(scrnsize(1)*.99, scrnsize(3)*.9, ['v_e' sprintf('=%6.2f
m/s',ve)], 'Color', 'y','FontName', 'Courier')
            text(scrnsize(2)*.5, scrnsize(3)*.8, ['w_i' sprintf('=%6.2f
m/s',wo)], 'Color', 'y','FontName', 'Courier')
            text(scrnsize(2)*.5, scrnsize(3)*.9, ['v_i' sprintf('=%6.2f
m/s',vo)], 'Color', 'y','FontName', 'Courier')
            text(scrnsize(2)*.5, scrnsize(3)*.7, ['t_i' sprintf('=%6.1f
min',tti/60)], 'Color', 'y','FontName', 'Courier')
        end
        if cntr>0, plot([0 0],[0 0],'*'), plot([0 0],[0 zoom], 'g'), plot([0
zoom],[0 0],'g', 'LineWidth', 2), end
        text(scrnsize(2)*.5, scrnsize(4)*.9, sprintf('z =%6.2f m',z(1)),
'Color', 'g','FontName', 'Courier')
        text(scrnsize(2)*.5, scrnsize(4)*.8, sprintf('y =%6.2f m',y(1)),
'Color', 'g','FontName', 'Courier')
        text(scrnsize(2)*.5, scrnsize(4)*.7, sprintf('w =%6.2f m/s',w(1)),
'Color', 'g','FontName', 'Courier')
        text(scrnsize(2)*.5, scrnsize(4)*.6, sprintf('v =%6.2f m/s',v(1)),
'Color', 'g','FontName', 'Courier')
        xlabel('y-axis (m)'), ylabel('OUT OF PLANE z-axis (m)')
    case 3 % XYZ Stationary
        plot3(y(1),x(1),z(1), 'cs'), hold on;
        if pdtr>0
            plot3(y,x,z, 'y:')
            text(y(end), x(end), z(end), sprintf('%0.1f min',pdtrl/60),
'Color', 'y','FontName', 'Courier')
        end
        if cntr>0
            plot3([0 0],[0 0],[0 -zoom*2], 'g')
            plot3([0 0],[0 zoom*2],[0 0], 'g')
            plot3([0 zoom*2],[0 0],[0 0], 'g', 'LineWidth', 2)
            plot3([0 0],[0 -zoom*2],[0 0], 'g:')
        end
        xlabel('y-axis (m)'), ylabel('x-axis (m)'), zlabel('z-axis (m)')
        scrnsize=[-1 1 -1 1 -1 1]*zoom*10;
        view(az,el)
        grid on
        set(gca,'XColor','b','YColor','b','ZColor', 'b')
    case 4 % XYZ Rotating
        Pdtr=R*[x;y;z];
        Px=Pdtr(1,:); Py=Pdtr(2,:); Pz=Pdtr(3,:);
        plot3(P(2,end),P(1,end),P(3,end), 'cs'), hold on;
        if pdtr>0
            plot3(Py,Px,Pz, 'y:')
            text(Py(end), Px(end), Pz(end), sprintf('%0.1f min',pdtrl/60),
'Color', 'y','FontName', 'Courier')
        end
        CTR=R*[0, 0, 0, 0, 0, 1, 0,-1;
            0, 0, 0, 1, 0, 0, 0, 0;
            0, -1, 0, 0, 0, 0, 0, 0]*zoom*2;
        if cntr>0
            plot3(CTR(2,1:2),CTR(1,1:2),CTR(3,1:2), 'g')
            plot3(CTR(2,3:4),CTR(1,3:4),CTR(3,3:4), 'g', 'LineWidth', 2)
            plot3(CTR(2,5:6),CTR(1,5:6),CTR(3,5:6), 'g')
            plot3(CTR(2,7:8),CTR(1,7:8),CTR(3,7:8), 'g:')
        end
```

```
        xlabel('Y_{ECI}-axis (m)'), ylabel('X_{ECI}-axis (m)'),
zlabel('Z_{ECI}-axis (m)')
        scrnsize=[-1 1 -1 1 -1 1]*zoom*10;
        view(az,el)
        grid on
        set(gca,'XColor','b','YColor','b','ZColor', 'b')
    case 5 % XYZ Rotating and Translated
        if zoom < .08*norm(Pt(:,end)), zoom = .1*norm(Pt(:,end)); end
        Pdtr=R*[x;y;z]+a*R(:,1)*ones(1,length(x));
        Px=Pdtr(1,:); Py=Pdtr(2,:); Pz=Pdtr(3,:);
        plot3(Pt(2,end),Pt(1,end),Pt(3,end), 'cs'), hold on;
        plot3(Pt(2,:),Pt(1,:),Pt(3,:), ':c')
        if pdtr>0
            plot3(Py,Px,Pz, 'y:')
            text(Py(end), Px(end), Pz(end), sprintf('%0.1f min',pdtrl/60),
'Color', 'y','FontName', 'Courier')
        end
        CTR=R*[0, 0, 0, 0, 0, 1, 0,-1;
            0, 0, 0, 1, 0, 0, 0, 0;
            0, -1, 0, 0, 0, 0, 0, 0]*zoom*2+a*R(:,1)*ones(1,8);
        if cntr>0
            plot3(CTR(2,1:2),CTR(1,1:2),CTR(3,1:2), 'g')
            plot3(CTR(2,3:4),CTR(1,3:4),CTR(3,3:4), 'g', 'LineWidth', 2)
            plot3(CTR(2,5:6),CTR(1,5:6),CTR(3,5:6), 'g')
            plot3(CTR(2,7:8),CTR(1,7:8),CTR(3,7:8), 'g:')
        end
        plot3(0,0,0,'*')
        xlabel('Y_{ECI}-axis (m)'), ylabel('X_{ECI}-axis (m)'),
zlabel('Z_{ECI}-axis (m)')
        scrnsize=[-1 1 -1 1 -1 1]*zoom*10;
        view(az,el)
        grid on
        set(gca,'XColor','b','YColor','b','ZColor', 'b')
    case 0 % Help Screen
        plot(0,0, 'cs'), hold on;
        text(1, -1,  'number pad: 1-9')
        text(2, -1,  'activate in-plane thrust')
        text(1, -2,  '+/-')
        text(2, -2,  '+/- out-of-plane thrust')
        text(1, -3,  '0')
        text(2, -3,  'stop relative motion')
        text(1, -4,  'shift-[ / ]')
        text(2, -4,  '+/- \DeltaV amount')
        text(1, -5,  'shift-1/2/3/4/5')
        text(2, -5,  'display mode')
        text(1, -6,  'p')
        text(2, -6,  'on/off predictor')
        text(1, -7,  '[ / ]')
        text(2, -7,  '+/- predictor length')
        text(1, -8,  'c')
        text(2, -8,  'on/off center mark')
        text(1, -9,  '*')
        text(2, -9,  'orbit in ellipse')
        text(1, -10, '.')
        text(2, -10, 'intercept')
        text(1, -11, '</>')
        text(2, -11, '+/- zoom')
        text(1, -12, ',')
        text(2, -12, 'auto-zoom')
        text(1, -13, '?')
        text(2, -13, 'help screen')
        text(1, -14, 'q')
        text(2, -14, 'quit')
```

```
        text(1, -18, '\DeltaV used thus far')
        text(2, -18, sprintf('%0.2f m/s',fuel), 'FontWeight', 'demi')
        axis([.9 3.1 -20 0])
        set(gca,'XTickLabel',[],'YTickLabel',[],'ZTickLabel',[])
        set(gca,'XTick',[],'YTick',[],'ZTick',[])
        xlabel('Help Screen')
end
title('CW Real Time Interactive Simulator')
if scrnmode < 3
    clck = clock; hr = clck(4); min = clck(5); sec = clck(6);
    text(scrnsize(1)*.99, scrnsize(4)*.9, sprintf('Time:
%2i:%02i:%02i',hr,min,round(sec)), 'Color', 'g','FontName', 'Courier')
    text(scrnsize(1)*.99, scrnsize(4)*.8, ['|\DeltaV| = ' sprintf('%0.2f m/s',
dV)], 'Color', 'g','FontName', 'Courier')
    text(scrnsize(1)*.99, scrnsize(4)*.7, ['a=' sprintf('%ikm, ', a) 'e=0'],
'Color', 'g','FontName', 'Courier')
end
if scrnmode, axis equal, axis(scrnsize), set(gca, 'Color', 'k'), end
frame = getframe(gca);
% aviobj = addframe(aviobj,frame);
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Romano, M., Friedman, D., & Shay, Tracy. (2005) Laboratory Experimentation of Autonomous Spacecraft Approach and Docking to a Collaborative Target. Accepted for publication. To Appear AIAA Journal of Spacecraft and Rockets.

2. Porter, M. (2002). Development and Control of the Naval Postgraduate School Planar Autonomous Docking Simulator (NPADS). MS Thesis, Dept. of Astro. and Mech. Eng., NPS.

3. Friedman, D. (2005). Laboratory Experimentation of Autonomous Spacecraft Docking Using Cooperative Vision Navigation. MS Thesis, Dept. of Astro. And Mech. Eng., NPS.

4. Shay, T.J. (2005). Design and Fabrication of Planar Autonomous Spacecraft Simulator with Docking and Fluid Transfer Capability. MS Thesis, Dept. of Astro. and Mech. Eng., NPS.

5. Hall, Jason. (2006). Design And Integration Of A Three Degrees-Of-Freedom Robotic Vehicle With Control Moment Gyro For The Autonomous Multi-Agent Physically Interacting Spacecraft (AMPHIS) Testbed. MS Thesis, Dept. of Astro. and Mech. Eng., NPS.

6. Price, Bill. (2006). Control System of a Three DOF Spacecraft Simulator by Vectorable Thrusters and Control Moment Gyros. MS Thesis, Dept. of Astro. and Mech. Eng., NPS.

7. Hall, Jason. (to be completed September 2009). PhD Dissertation, Dept. of Astro. and Mech. Eng., NPS.

8. McCamish, Shawn. (to be completed December 2007). Distributed Autonomous Control of Multiple Spacecraft in Close Proximity Operations. PhD Dissertation, Dept. Elec. And Comp. Eng., NPS.

9. Styliadis, A. D., Karagiannidis, C. J., Zestas, N. C., Chatzara, K.N., Pose Estimation from a Single Photograph in a Controlled CAD Environment.

10. Vallado, David A., Fundamentals of Astrodynamics and Applications, Microcosm Press, 2001

11. Alfriend, Terry, Notes on Relative Motion of Neighboring Satellites, NPS, 2006

12. Newman, Jim, Lectures on RPOP and the Space Shuttle/ISS rendezvous, NPS, 2006.

13. SICK. (2004). LD Laser Scanners Product Information.

14. Yakimenko, O., Direct method for Rapid Prototyping of Near-Optimal Aircraft Trajectories, AIAA Journal of Guidance, Control, and Dynamics, 23(**5**), 2000, pp.865-875.

15. Yakimenko, O.A., Kaminer, I.I., Lentz, W.J., Ghyzel, P.A., Unmanned Aircraft Navigation for Shipboard Landing Using Infrared Vision, IEEE Transactions on Aerospace and Electronic Systems, 38(**4**), 2002, pp.1181-1200.

16. Yakimenko, O.A., Dobrokhodov, V.N., Kaminer, I.I., Berlind, R.M., Autonomous Scoring and Dynamic Attitude Measurement, Proceedings of the 18th AIAA Aerodynamic Decelerator Systems Technology Conference and Seminar, Munich, Germany, May 24-25, 2005.

17. Eikenberry, B. D., Yakimenko, O., Romano, M., A Vision Based Navigation among Multiple Flocking Robots: Modeling and Simulation. Proceedings of the AIAA Modeling and Simulation Technologies Conference, Keystone, Colorado, August 22nd, 2006.

18. Ross, I. M., Class notes from Astrodynamic Optimization, AE4850. NPS, 2006.

19. NASA. DARTing Into Space, http://www.nasa.gov/missions/science/dart_into_space.html.

20. Weismuller, T., Leinz, M., GN&C Technology Demonstrated by the Orbital Express Autonomous Rendezvous and Capture Sensor System. Preceedings from the 29th Annual AAS Guidance and Control Conference, Breckenridge, CO, Feb. 4-8, 2006.

21. Sapce Daily. ETS-7 – Orbital Rendezvous and Robotic Mission. http://www.spacedaily.com/spacenet/text/ets7-b.html. Tokyo, Apr. 2, 1997.

22. ESA. Successful tests of ATV rendezvous replicate the 2007 Jules Verne mission. http://www.esa.int/SPECIALS/ATV/SEMWCEKKKSE_0.html.

23. Banke, K., Air Force XSS-10 Micro-Satellite Mission a Success. http://www.space.com/missionlaunches/xss10_update_030130.html. Cape Canaveral Bureau, Jan. 30, 2003.

24. Mitchell, I. T., Gorton, T. B., Taskov, K., Drews, M. E., Luckey, D., Osborne, M. L., Page, L. A., Norris, H, L. III, Shepperd, S. W., GN&C Development of the XSS-11 Micro Satellite for Autonomous Rendezvous and Proximity Operations. Preceedings from the 29th Annual AAS Guidance and Control Conference, Breckenridge, CO, Feb. 4-8, 2006.

25. Miotto, P., Zimpfer, D., Mamich, H., Autonomous Mission Manager And GN&C Design for the Hubble Robotic Vechile De-Orbit Module. Preceedings from the 29th Annual AAS Guidance and Control Conference, Breckenridge, CO, Feb. 4-8, 2006.

26. The CX OLEV Space Tug. http://www.orbitalrecovery.com/cxolev.htm.

27. Bosse, A. B., Barnds, W. J., Brown, M. A., Creamer, N. G., Feerst, A., Henshaw, C. G., Hope, A. S., Kelm, B. E., Klein, P. A., Pipitone, F., Plourde, B. E., Whalen, B. P., SUMO: Spacecraft for the universal modification of orbits. Procedings of SPIE Vol. 5419, Bellingham, WA, 2004.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Marcello Romano
   Naval Postgraduate School
   Monterey, California

4. Jim Newman
   Naval Postgraduate School
   Monterey, California